uc3m | Universidad Carlos III de Madrid

Grado Universitario en Tecnologías Industriales

2017 – 2018

Trabajo Fin de Grado

CONTROL DEL MOVIMIENTO EN MOTORES ELÉCTRICOS PARA ROBOT SOCIAL

Jorge Bustos Sánchez

Tutor

Fernando Alonso Martín

25 de septiembre del 2018, Madrid

RESUMEN

Se denomina Robot Social, a aquella entidad mecánica que se comunica e interactúa con seres humanos. El objetivo de un Robot Social, es realizar esta interacción máquina-humano tan natural, como aquella que establecemos los seres humanos unos con otros. Hay numerosos parámetros que influyen en la naturalidad de la interacción máquina-hombre. Uno de los más importantes, es el movimiento del robot social, parámetro hacia el cual se enfoca este proyecto.

El ser humano, a menudo, acompaña su diálogo con gestos y movimientos de brazos, cabeza, manos, etc. Estos movimientos, se caracterizan por ser fluidos y variados. Debido a que la interacción máquina-hombre, busca asemejarse a la existente entre seres humanos, los robots sociales deben desempeñar movimientos similares a los nuestros.

En este trabajo, se plantea la mejora de una API, del "Proyecto Mini" del Laboratorio de Robótica Social de la Universidad Carlos III de Madrid, que inicialmente realizaba gestos y movimientos a una velocidad constante y siempre fija. Esta mejora no sólo permite el desempeño de gestos y movimientos a velocidades variables y configurables, permitiendo una interacción hombre-máquina natural y fluida, sino también la obtención de parámetros y realización de acciones de los motores encargados del movimiento del robot.

Palabras clave: API, ROS, control, motor, dynamixel.

AGRADECIMIENTOS

Deseo expresar mi agradecimiento a las personas que me ha ayudado en el desarrollo de este trabajo, sobre todo a mi tutor Fernando Alonso, que a pesar de su ajetreada agenda siempre ha estado ahí para ayudarme; a mis padres Miguel Ángel Bustos y María Del Carmen Sánchez, mi hermana, mi novia y mis amigos que me han apoyado en los momentos de mayor frustración y me han sufrido durante estos meses de duro trabajo. A ellos les dedico este trabajo. Mi agradecimiento así mismo a todos los involucrados en el Laboratorio de Robótica Social de la Universidad Carlos III por la oportunidad y por la ayuda.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

En la presente memoria, se describe el Trabajo de Fin de Grado que tiene como objetivo la implementación de una API de control de los motores eléctricos del robot "Mini".

La realización del proyecto se ha llevado a cabo a través del sistema operativo Linux 14.04 y ROS Indigo en lenguaje C++, cuya utilidad presenta un paso adelante en el desarrollo del "Proyecto Mini" por parte del Laboratorio de Robótica Social de la Universidad Carlos III.

1.1 Motivación

La robótica es una disciplina que está creando un gran impacto en el mundo de la industria. Un ejemplo, son los grandes almacenes automatizados en China que son capaces de gestionar 200.000 pedidos al día con la ayuda de tan solo 4 humanos. Pero la "robótica", es una palabra muy genérica que engloba numerosos campos. En este trabajo, en concreto, se trata la robótica social y su impacto en la vida de las personas.

La robótica social no es una disciplina nueva. ASIMO, el famoso robot humanoide creado por Honda, abrió el camino en el año 2000 de una tecnología que está en continuo proceso de investigación y que ha llegado recientemente a nuestros hogares con el robot social conocido como JIBO.

Figura 1: Robot ASIMO. [10]

Figura 2: Robot JIBO. [7]

Es por esta importancia que tiene la robótica en la industria y la que tiene y tendrá en la vida cotidiana del ser humano, la principal motivación del desarrollo de este trabajo; así como la participación en un proyecto de relevancia para la Universidad Carlos III de Madrid como es el "Proyecto Mini".

1.2 Objetivo

Los objetivos del trabajo pueden distinguirse en dos tipos: aquellos necesarios para el correcto funcionamiento del proyecto y los perseguidos por el alumno como consecuencia de su realización.

En cuanto a los objetivos necesarios para el correcto funcionamiento del trabajo, el objetivo incial, iba enfocado a la realización de un algoritmo capaz de generar movimientos a velocidad variable en el robot "Mini", para poder implementarlos en el proyecto.

Una vez realizado el algoritmo, se dio paso a la realización del objetivo final: una API de control y gestión de los motores del robot del "Proyecto Mini", capaz de gestionar los movimientos, los parámetros y la calibración de los mismos.

El "Proyecto Mini" o "Proyecto MiniMaggie", llevado por el Laboratorio de Robótica Social de la UC3M, es un proyecto secuela del afamado robot social Maggie. Maggie, es un robot social orientado a la interacción con niños. Este, es capaz de moverse por el entorno con autonomía, evitar obstáculos, conectarse a Internet y dar información accesible. Sin embargo, su así llamada hermana MiniMaggie o Mini tiene un fin distinto. Se trata de un peluche robotizado, de unos 35 centímetros de altura, capaz de interaccionar y expresar emociones básicas con fines terapéuticos. Actualmente, Mini se encuentra operativa en una clínica en Zamora, como parte de un programa de la UC3M con la Fundación Alzheimer España, aunque sigue en continuo desarrollo. [14] [8]

Los objetivos marcados por parte del alumno en la realización del proyecto consistían en primer lugar, en el correcto aprendizaje de los lenguajes de programación utilizados en el proyecto y, en segundo lugar, en la compresión y obtención de conocimientos de los mecanismos de gestión y realización de proyectos de robótica en el campo de la investigación. Todo ello, ha permitido al alumno tener una buena base de conocimientos en robótica para poder aplicarlos en futuros proyectos.

Figura 3: Imagen del robot Maggie. [14]

Figura 4: Imagen del robot Mini. [5]

1.3 Estructura

El principal objetivo de este documento, es mostrar y explicar el código de la API para el control de movimientos en motores eléctricos para el robot mini. Para ello, es necesario contextualizar al lector. Primero, en el **capítulo 1**, el documento presenta una introducción donde se expresan los objetivos y motivaciones del trabajo, en el **capítulo 2**, se introduce brevemente a la materia mediante el estado del arte, donde se explica la historia de la robótica así como las principales tecnologías y motores empleados. En el **capítulo 3,** se introduce al trabajo mediante la explicación de las principales tecnologías utilizadas en él.

Una vez que el lector este contextualizado, se da inicio a la resolución del trabajo en el **capítulo 4**. Se realiza un planteamiento del problema, y un planteamiento y desarrollo de la solución dividido en dos fases por las que ha pasado el trabajo. En el **capítulo 5**, se realiza una exposición de los resultados obtenidos y una conclusión respecto a ellos en el **capítulo 6**. Por último, en el **capítulo 7**, se incluyen las líneas futuras de trabajo del TFG. Además, en el **capítulo 8**, para facilitar la comprensión de la API se introduce un manual de uso, donde se detallan los comandos a introducir para cada acción que se desea realizar.

2 ESTADO DEL ARTE

En este capítulo, se introduce al lector en la robótica social. Se realizará un breve resumen sobre su historia y evolución, se analizarán sus funciones y las tecnologías empleadas en ella, así como los principales tipos de motores usados en la robótica social.

2.1 Historia de la robótica social

La robótica, es una disciplina que presenta cierta ambigüedad respecto a su nacimiento. Se puede considerar a Leonardo Da Vinci como su inventor, quien a lo largo del siglo XV se encargó de diseñar un robot humanoide, o incluso a un ingeniero mecánico del Rey Mu de Zhou en el siglo III a.C.; pero el término "robótica" nace de manos de Isaac Asimov, después de que en 1923 Karel Čapek en su obra "Rossum´s Universal Robots / R.U.R." acuñase la palabra "Robot" de la palabra checa "*robota"*, que significa servidumbre o trabajo forzado. [4]

Sin embargo, la robótica social que conocemos no nacería hasta 1986, cuando Honda inicia su programa de I+D que tiene como objetivo la coexistencia y cooperación de los robots con los seres humanos. Este programa, daría lugar al primer robot humanoide y social ASIMO en el año 2000. [6]

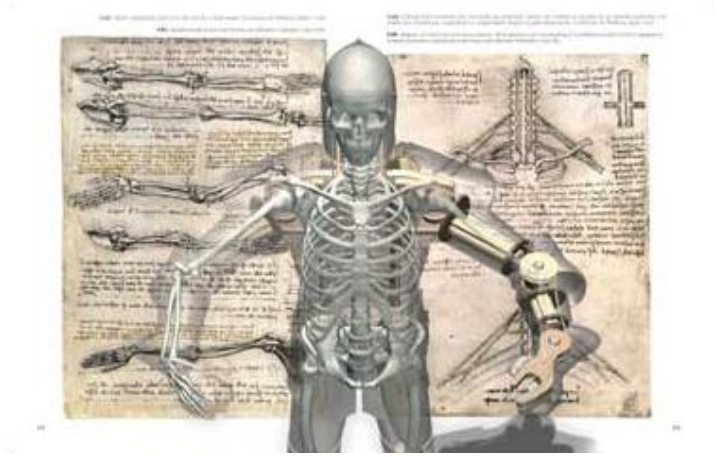

Figura 5: Imagen del robot prototipo de Leonardo Da Vinci. [9]

2.2 Tecnologías empleadas

La robótica social, como ya se ha dicho anteriormente, persigue la interacción de los robots con la sociedad humana. Para que esto sea posible, se emplean numerosas tecnologías tanto en la parte de hardware como en la de software.

- **Hardware**: Una de las bases en la robótica, son los sensores. Los sensores dotan a los robots de la capacidad de percibir su entorno y de recabar una gran cantidad de información. Esta capacidad de percepción del entorno y de su información es la misma que utilizan los seres humanos en su interacción cotidiana y que les conduce a una determinada conducta en función de su entorno. Por ejemplo, un ser humano no interacciona igual con una persona que está llorando que con una persona que está sonriendo. Al igual que no lo hace un ser humano, un robot social tampoco debe hacerlo. Toda esta información percibida a través de los sensores, es la que le permite, mediante distintos programas, la capacidad de tomar decisiones y dotar de experiencia a un robot. Otro elemento básico, que es la base de este trabajo son los actuadores, en concreto, motores. Se hablará más delante de ellos.

- **Software**: El software es la otra parte fundamental en el desarrollo de la robótica social. Una de las tecnologías más determinantes durante los últimos años en la rama de la inteligencia artificial, ha sido la gestión de toma de decisiones mediante sistemas motivacionales. Estos sistemas son una solución alternativa a los sistemas jerárquicos (sistemas donde los objetivos estaban marcados por jerarquías y preprogramados) que basan la realización de una actividad u otra por parte del robot en función sus necesidades. [13]

  Antes de la invención de tecnologías de inteligencia artificial, surgieron entornos de trabajo capaces de gestionar un sistema tan complejo como lo es un robot. Entre los más utilizados en la actualidad se encuentra ROS. Se trata de un sistema operativo creado en 2007 en la Universidad de Stanford como parte del proyecto STAIR. Se puede decir que es la arquitectura más influyente en el diseño y desarrollo de robots.

2.3 Motores en la robótica social

Como se ha comentado en el apartado anterior los actuadores en la robótica social son motores. Hay distintos tipos de motores. Para la elección del tipo de motor hay varias propiedades a tener en cuenta en función de su aplicación, como su potencia, precisión, velocidad, peso, volumen y coste. Teniendo en cuenta estas propiedades se puede elegir entre los siguientes tipos de motores:

- **Motores de corriente continua**: el movimiento de estos motores se basa en el desfase proporcionado por dos campos magnéticos, uno creado por el estator (imán fijo del motor) y otro creado por la corriente que recorre el circuito eléctrico interior del motor. Cuando ambos campos magnéticos se alinean se produce la parada del motor.

  Estos tipos de motores pueden ser de distintos tamaños, en función de la potencia transmitida que se desea. Poseen altas velocidades de giro y bajo par. Sin embargo, el control de la posición y la velocidad es una de sus desventajas debido a su comportamiento no lineal.

  Una variante de los motores de corriente continua, son aquellos que contienen reductores. Estos se suelen utilizar para el arranque y accionamiento de robots y vehículos.

- **Motores brushless**: los motores brushless a diferencia de los motores de corriente continua el desfase de campos magnéticos no se realiza mediante corriente eléctrica, sino que se recurre a la electrónica para la inversión de la corriente. Esta diferencia permite que estos motores adquieran mayores velocidades, tengan menos peso y mayor durabilidad que los motores de corriente continua tradicionales. Su principal aplicación son los vehículos aéreos como los drones.

- **Servomotores**: estos motores requieren para su movimiento del envío de la instrucción deseada, la cual es gestionada por un procesador que transmite la instrucción para su movimiento. Este tipo de motores no pueden dar vueltas completas. Sin embargo, el control de posición y giro son de muy alta precisión. Este tipo de motores son los más utilizados en los proyectos de robótica, en concreto, para el movimiento de articulaciones.

- **Servomotores de rotación continua:** los servomotores de rotación continua son una variante de un servo convencional. Tal y como indica su nombre estos motores permiten dar una vuelta completa, al igual que un motor de corriente continua; con el aditivo que hay un control de la velocidad.
- **Motores paso a paso:** estos motores son dispositivos electromagnéticos y rotativos conversores de pulsos digitales en movimiento mecánico. La posición de rotación es directamente proporcional al número de pulsos y la velocidad a la frecuencia de estos. Estos motores son otro de los tipos más empleados en proyectos de robótica, debido a su precisión, bajo mantenimiento y coste. [12]

3 TECNOLOGÍAS ASOCIADAS

En este capítulo se procede a la descripción de las tecnologías usadas para el desarrollo del trabajo. En primer lugar, se hablará brevemente de los motores empleados (Dynamixel AX-12), y en segundo lugar del sistema operativo utilizado (ROS) y del lenguaje de programación utilizado (C++), así como del paquete del que se parte para la realización de la API.

3.1 Servomotor dynamixel AX-12

El servomotor dynamixel AX-12 es un actuador inteligente desarrollado y diseñado para estructuras robóticas. A lo largo del trabajo se harán numerosas referencias a los motores dynamixel AX-12. Con el fin de optimizar y abreviar la lectura se hará referencia a los motores dynamixel AX-12 mediante la palabra “motor/motores”.

3.1.1 Especificaciones

- Dimensiones: 32mm X 50mm X 40mm.
- Peso: 53.5g.
- Material: Plástico ingenieril.
- Resolución: 0.29º.
- Señal de comunicación: paquete digital.
- Tipo de protocolo de comunicación: “Half dúplex Asynchronous Serial Communicaction”.

3.1.2 Tabla de Control

El motor dynamixel AX-12 presenta una tabla de control que permite al usuario controlar el motor mediante el acceso a varios registros instalados en una memoria EEPROM y RAM.

Los valores de los datos de los distintos registros funcionan de diversa manera para la RAM y EEPROM. Los datos de la RAM se fijan al valor inicial cuando la fuente de alimentación se enciende, mientras que los datos de la EEPROM no son volátiles, permaneciendo así fijados cuando la fuente de alimentación este apagada.

Tabla 1: Tabla de registros de la memoria EEPROM de los motores dynamixel AX-12. [2]

	Address	Item	Access	Initial Value
EEPROM Area	0(0X00)	Model Number(L)	RD	12(0x0C)
	1(0X01)	Model Number(H)	RD	0(0x00)
	2(0X02)	Version of Firmware	RD	?
	3(0X03)	ID	RD,WR	1(0x01)
	4(0X04)	Baud Rate	RD,WR	1(0x01)
	5(0X05)	Return Delay Time	RD,WR	250(0xFA)
	6(0X06)	CW Angle Limit(L)	RD,WR	0(0x00)
	7(0X07)	CW Angle Limit(H)	RD,WR	0(0x00)
	8(0X08)	CCW Angle Limit(L)	RD,WR	255(0xFF)
	9(0X09)	CCW Angle Limit(H)	RD,WR	3(0x03)
	10(0x0A)	(Reserved)	-	0(0x00)
	11(0X0B)	the Highest Limit Temperature	RD,WR	85(0x55)
	12(0X0C)	the Lowest Limit Voltage	RD,WR	60(0X3C)
	13(0X0D)	the Highest Limit Voltage	RD,WR	190(0xBE)
	14(0X0E)	Max Torque(L)	RD,WR	255(0XFF)
	15(0X0F)	Max Torque(H)	RD,WR	3(0x03)
	16(0X10)	Status Return Level	RD,WR	2(0x02)
	17(0X11)	Alarm LED	RD,WR	4(0x04)
	18(0X12)	Alarm Shutdown	RD,WR	4(0x04)
	19(0X13)	(Reserved)	RD,WR	0(0x00)
	20(0X14)	Down Calibration(L)	RD	?
	21(0X15)	Down Calibration(H)	RD	?
	22(0X16)	Up Calibration(L)	RD	?
	23(0X17)	Up Calibration(H)	RD	?

Tabla 2: Tabla de los registros de la memoria RAM de los motores dynamixel AX-12. [2]

RAM Area	24(0X18)	Torque Enable	RD,WR	0(0x00)
	25(0X19)	LED	RD,WR	0(0x00)
	26(0X1A)	CW Compliance Margin	RD,WR	0(0x00)
	27(0X1B)	CCW Compliance Margin	RD,WR	0(0x00)
	28(0X1C)	CW Compliance Slope	RD,WR	32(0x20)
	29(0X1D)	CCW Compliance Slope	RD,WR	32(0x20)
	30(0X1E)	Goal Position(L)	RD,WR	[Addr36]value
	31(0X1F)	Goal Position(H)	RD,WR	[Addr37]value
	32(0X20)	Moving Speed(L)	RD,WR	0
	33(0X21)	Moving Speed(H)	RD,WR	0
	34(0X22)	Torque Limit(L)	RD,WR	[Addr14] value
	35(0X23)	Torque Limit(H)	RD,WR	[Addr15] value
	36(0X24)	Present Position(L)	RD	?
	37(0X25)	Present Position(H)	RD	?
	38(0X26)	Present Speed(L)	RD	?
	39(0X27)	Present Speed(H)	RD	?
	40(0X28)	Present Load(L)	RD	?
	41(0X29)	Present Load(H)	RD	?
	42(0X2A)	Present Voltage	RD	?
	43(0X2B)	Present Temperature	RD	?
	44(0X2C)	Registered Instruction	RD,WR	0(0x00)
	45(0X2D)	(Reserved)	-	0(0x00)
	46[0x2E)	Moving	RD	0(0x00)
	47[0x2F)	Lock	RD,WR	0(0x00)
	48[0x30)	Punch(L)	RD,WR	32(0x20)
	49[0x31)	Punch(H)	RD,WR	0(0x00)

3.1.3 Protocolo de comunicación

La comunicación se realiza a través de dos tipos de paquetes, el "Paquete de Instrucción" enviado por el controlador principal al servomotor, y el "Paquete de Estado" enviado por el servomotor al controlador principal en respuesta al primero, tal y como refleja la siguiente imagen:

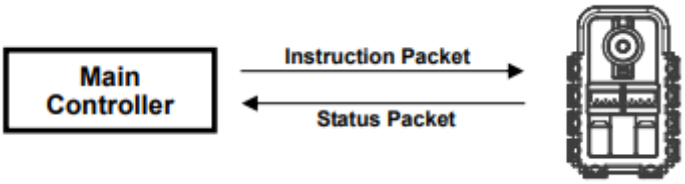

Figura 6: Esquema de comunicación entre el controlador y el motor. [2]

Ambos paquetes se caracterizan por estar formados por 8 bits, 1 bit de parada y sin paridad. Sin embargo, presentan diferencias entre sí.

3.1.3.1 Paquete de Instrucción

El Paquete de Instrucción presenta la siguiente estructura:

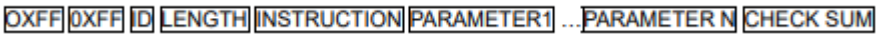

Figura 7: Estructura del paquete de instrucción. [2]

- OXFF | OXFF: indican el comienzo de un paquete entrante.
- ID: el número de identidad del motor Dynamixel al que se le envía la instrucción. Cada motor tiene un ID asociado comprendido entre 0 y 254. Este dato del paquete, permite identificar el motor al que va dirigido el paquete de instrucción o del que se recibe el paquete de estado (véase figura

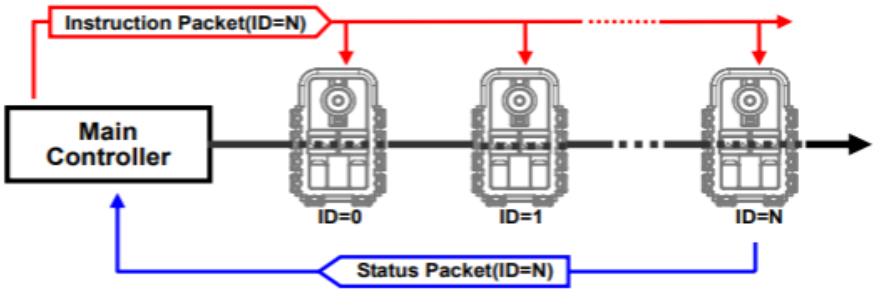

Figura 8: Comunicación para N motores. [2]

- LENGTH: la longitud del paquete.
- INSTRUCTION: la instrucción que se desea que ejecute el motor.
- PARAMETER1…N: utilizados como espacio adicional para enviar otra instrucción en el caso de que fuera necesario.
- CHECK SUM: bit de verificación de la estructura del paquete.

3.1.3.2 Paquete de Estado

El Paquete de Estado presenta la siguiente estructura:

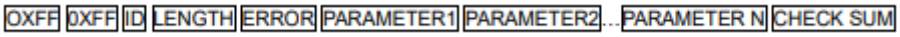

Figura 9: Estructura del paquete de estado. [2]

La estructura es muy similar al del Paquete de Instrucción, el único elemento que cambia es INSTRUCTION por ERROR, que envía distintos avisos de error por parte de la unidad Dynamixel.

3.2 ROS (Robotic Operating System) y C++

ROS (Robotic Operating System) o Sistema Operativo Robótico traducido al español, es un sistema operativo tal y como indica su nombre que establece un entorno de trabajo con una filosofía de desarrollo y programación capaz de aunar las necesidades para la gestión de un sistema robótico. Se caracteriza por:

- Presentar una estructura de funcionamiento de grafos.
- Ser Open-source, es decir que la mayoría del software es público.
- Capacidad de multilenguaje. Los distintos módulos de ROS permiten escritura en cualquier lenguaje de programación (C++, Python, Java, etc.). En este trabajo se utiliza en concreto el C++. [11]

3.3 Paquete Dynamixel_motor

Para la realización de este trabajo el alumno se ha basado en la utilización de un paquete creado por arebgun [3] (se trata de un third-party), que contiene un nodo configurable, servicios y una secuencia de herramientas para iniciar, parar y reiniciar uno o más servomotores de dynamixel. El paquete contiene los siguiente:

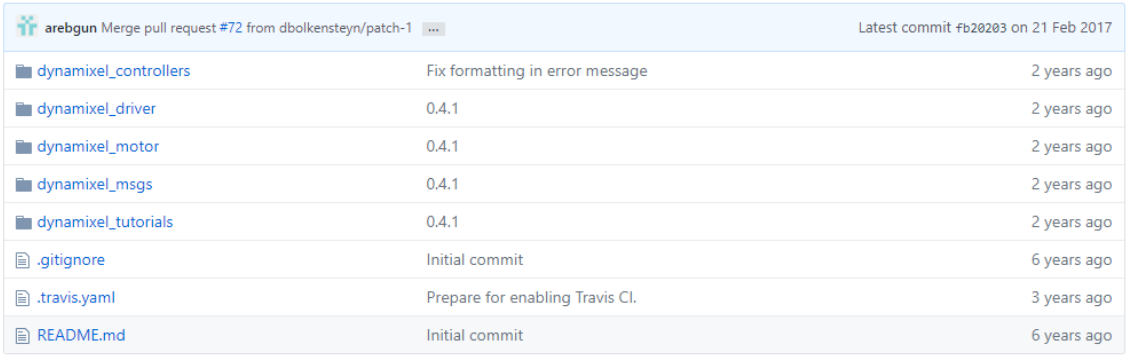

Figura 10: Estructura del paquete dynamixel_motor. [3]

A parte del nodo principal, el paquete ofrece diversos mensajes y servicios al usuario para su utilización con motores Dynamixel.

Se ha elegido este paquete como base para el desarrollo del trabajo debido a su conocimiento de uso y funcionamiento, y a la simplificación del trabajo para el desarrollo de la API. De no haber elegido este camino de desarrollo el alumno debería haber desarrollado la API a partir de un nivel más bajo creando funciones de lectura y escritura mediante los paquetes de instrucción y estado mencionados anteriormente.

4 DISEÑO Y DESARROLLO DE LA SOLUCIÓN

En este capítulo se expone el procedimiento de resolución tomado para la realización del TFG, partiendo de los obstáculos iniciales, hasta la consecución de la solución final. Primero se realizará un análisis general del problema a resolver, que permite dar al lector, una visión general de los obstáculos a afrontar. A continuación, después de haber diferenciado el desarrollo del TFG en dos partes, se planteará la solución y se desarrollará para ambas partes.

4.1 Planteamiento del problema

Antes de ponerse manos a la obra con la realización del trabajo, es necesario hacer un análisis global de los requisitos que hay que cumplir. Los **requisitos** principales son los **funcionales** (véase la tabla 3), son aquellos necesarios para que se realicen las funciones que se quieren dar a los motores. Sin embargo, a parte del cumplimiento de estos requisitos, es necesario el cumplimiento de unos **requisitos no funcionales.** Los requisitos no funcionales, por otro lado, consisten en las características de funcionamiento del sistema, como de su calidad. Para cumplir estos últimos, es necesario que el consumo de CPU y de memoria sea el mínimo posible.

Tabla 3: Requisitos funcionales de la API.

		Descripción
Entradas	**Activar**	Permite habilitar el movimiento del robot por una fuerza externa.
	Desactivar	Permite deshabilitar el movimiento del robot por una fuerza externa.
	Ping	Verifica si el motor se encuentra encendido.
	Comandar	Realiza el movimiento de un motor determinado, permitiéndole dar posición, velocidad y aceleración.
	Mover a posición inicial	Devuelve todos los motores a la posición inicial predefinida.

Salidas	**Obtener Estado**	Lee el estado actual del motor: posición, velocidad, temperatura, voltaje, etc.

Sin embargo, estos requisitos funcionales no aparecen como objetivo desde un principio. El objetivo inicial del TFG, era la realización de un algoritmo capaz de realizar movimientos a velocidades distintas y variables. Una vez conseguido, se dio paso al desarrollo de una API que cubriese las necesidades descritas en la tabla. Debido a este cambio de objetivos en la realización del trabajo, se explica el planteamiento del problema y de la solución en dos fases distintas.

4.1.1 Fase 1: Realización de movimientos a velocidades distintas y variables

Tal y como se ha comentado anteriormente, el principal el objetivo de esta primera fase del trabajo consiste en realizar movimientos a velocidades distintas y variables. Para ello el estudiante partía de una API existente e inacabada para la realización de movimientos en los motores, realizada por un exinformático del Laboratorio de Robótica Social de la UC3M.

Debido a esto, los principales problemas a afrontar son:

- Primero, la comprensión el sistema operativo robótico ROS, fundamental para la comprensión y desarrollo del trabajo.
- Segundo, el entendimiento del funcionamiento de los motores, para poder realizar el movimiento a velocidades distintas y variables a través de ROS.
- Y, por último, comprender el funcionamiento de la API existente, para entender al completo la comunicación de ROS con los motores.

4.1.2 Fase 2: Realización de una API de gestión y control de los motores del "Proyecto Mini"

Una vez resuelta la primera fase del trabajo, el alumno se enfrenta a la realización de una API capaz de gestionar y controlar los motores del "Proyecto Mini", con las funcionalidades mostradas en la tabla 3.

Para ello, el alumno se enfrenta a un problema más complejo, donde tiene que aprender a programar desde cero, de manera óptima y comprensible, con el fin de que sea entendible para los futuros desarrolladores y usuarios que la utilicen.

4.2 Planteamiento y desarrollo de la solución

En este apartado se afrontará primero el planteamiento de la solución según el análisis realizado en el planteamiento del problema y posteriormente el desarrollo de la solución. Se realizará de ambas fases por separado.

4.2.1 Fase 1: Realización de movimientos a velocidades distintas y variables

Primero se realizará un **planteamiento de la solución.** El alumno en este primer problema se enfrenta a un código en ROS, en lenguaje C++ ya existente, que permite el movimiento de los motores siempre a la misma velocidad.

El primer paso consiste en el estudio y comprensión del funcionamiento de ROS. Para ello el alumno ha estudiado su funcionamiento a través de los numerosos tutoriales y explicaciones existentes en Internet tanto por parte de la principal fuente de ayuda para la resolución de dudas de ROS, como es wiki-ros[1], como por parte de los distintos seminarios online que ofrecen diversas universidades.

Una vez asimilado y comprendido el funcionamiento de ROS se procede al análisis del código ya existente:

- Primero, se realiza una lectura del código para entender su finalidad, a través de la cual, se concluye que tiene como fin, la realización de una futura API para el control de los motores.
- El siguiente paso, es ejecutar el código existente para observar su funcionamiento. Para ello, se lanza el *launcher* encargado del movimiento de los motores.
- Después, se publica el *topic* encargado del movimiento del motor, en el cual hay que introducir la posición a la que se desea mover el motor. Mediante el comando "*rostopic pub*", y el nombre del *topic* encargado del movimiento del motor.

[1] http://wiki.ros.org/es

- Una vez realizadas estas operaciones, se procede a utilizar el comando "*rqt_graph*", que permite visualizar cómo interaccionan los nodos y los topics entre sí. Con este comando se obtiene la siguiente gráfica:

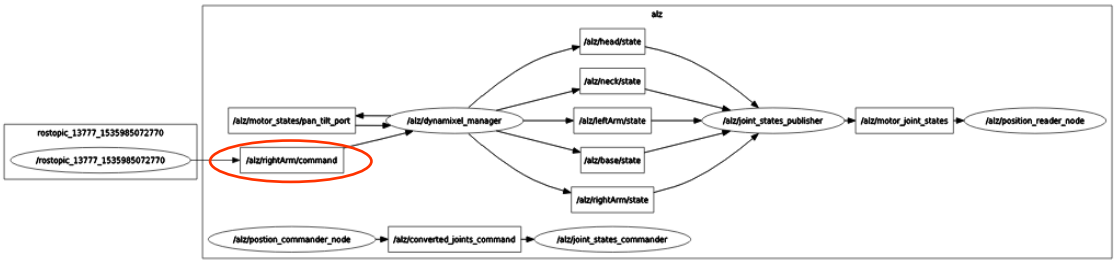

Figura 11: Gráfica que representa las conexiones entre los distintos *topics* y nodos que se ejecutan al lanzar el *launcher*.

Se puede observar en la gráfica, como el nodo "*/alz/dynamixel_manager*", se encuentra suscrito al *topic* publicado *"/alz/rightArm/command*" y realiza varias publicaciones a las que los demás nodos están suscritos. Para profundizar más y ver qué nodo realmente este realizando el movimiento se necesita más información de cada nodo. Para ello se utiliza el comando "*rosnode info*" y el nombre del nodo del que queremos obtener nuestra información. Una vez realizada la operación sobre todos los nodos, se va a centrar la atención en el nodo "*/alz/dynamixel_manager*". Se aplica el comando "*$rosnode info /alz/dynamixel_manager*" mediante el cual se obtiene lo siguiente:

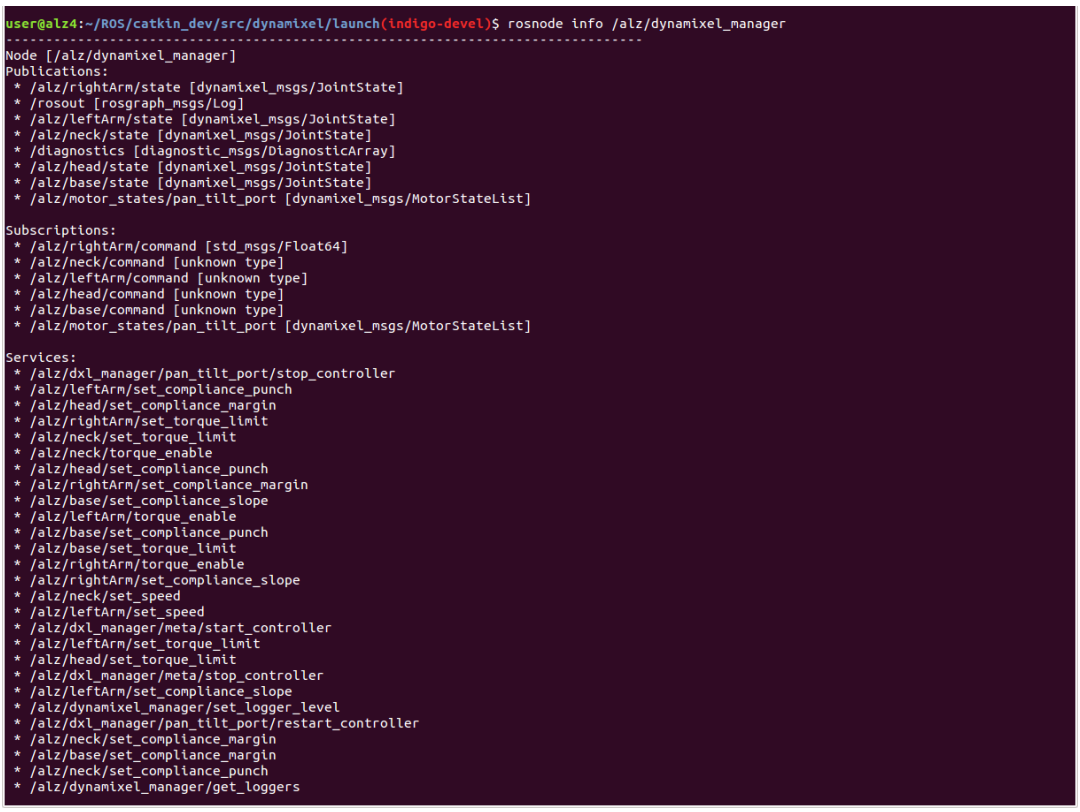

Figura 12: Listado de las publicaciones, subscripciones y servicios asociados con el nodo *„/alz/dynamixel_manager“.*

En la imagen se muestran los subscriptores, las publicaciones y los servicios del nodo “*/alz/dynamxiel_manager*”. Se puede observar que este se subscribe a varios *topics*, de los cuales son interesantes cinco; los cinco *topics* encargados del movimiento de los cinco motores existentes en el robot, y que reciben el nombre de *base*, *rightArm*, *leftArm*, *neck* y *head*. De ahí las suscripciones “*/alz/base/command*”, “*/alz/rightArm/command*”, etc.

Habiendo comprobado que el nodo “*/alz/dynamixel_manager*”, es el encargado de realizar el movimiento de los motores, y sabiendo, que los motores son capaces de cambiar la velocidad, ya que existen registros en la RAM y EEPROM encargados de la gestión de la velocidad de los motores, hay que centrarse en este nodo y en como comunicarse con él para cambiar la velocidad.

Una vez identificado y enfocado el camino de la solución, ya se puede dar comienzo el **desarrollo de la solución**.

Para ello, primero se investiga sobre el origen del nodo "*/alz/dynamixel_manager*". Tras una lectura y búsqueda de información, se llega a la conclusión que es el nodo principal de gestión de los motores Dynamixel del paquete *dynamixel_motor* creado por arebgun, mencionado en el apartado 3.3.

De vuelta al nodo "*/alz/dynamixel_manager*", es necesario detenerse en las subscripciones y servicios que ofrece, con el fin de resolver la comunicación entre el usuario y el motor para cambiar la velocidad. Para ello, hay que fijarse en la figura 11, aquí se observa un gran número de servicios entre los cuales hay cinco llamados "*/alz/base/set_speed*", "*/alz/rightArm/set_speed*", "*/alz/leftArm/set_speed*", "*/alz/neck/set_speed*" y "*/alz/head/set_speed*". Debido a la similaridad estructural de los nombres, con los cinco *topics* encargados del movimiento de los motores, y los nombres de los servicios ("*set_speed*"), se procede a ejecutarlos.

Para ello, se vuelve a lanzar nuestro documento *launcher*, y se llama al servicio deseado, por ejemplo "*/alz/head/set_speed*". Para llamar al servicio se tiene que ejecutar el siguiente comando: *"$rosservice call /alz/head/set_speed"* y se introduce el valor de velocidad deseado. Una vez introducido, se ejecuta el comando de movimiento del motor "*$rostopic pub /alz/head/command*" y el valor de la posición deseada.

Tras varias pruebas insertando distintas velocidades, se puede deducir por el cambio de velocidad observado de los motores en las diversas pruebas, la variación de velocidad en él.

4.2.2 Fase 2: Realización de una API de gestión y control de los motores del "Proyecto Mini"

Al igual que con la fase uno, es necesario hacer un previo análisis al desarrollo de la solución realizando un **planteamiento de la solución**.

Debido a las posibilidades que ofrece el paquete *dynamixel_motor*, se plantea desarrollar un API que se sitúe un nivel por encima, en el flujo de comunicación, de este paquete (véase figura 12).

Figura 13: Esquema a nivel global del flujo de comunicación de la API con los motores.

Tal y como se ha mostrado en el planteamiento del problema, la API consta de las funcionalidades mostradas en la tabla 3. Para ofrecer las funcionalidades descritas, es necesario plantearse el esquema de comunicación que se quiere seguir (*topics* o servicios), para cubrir cada una de las funcionalidades. Tras pensar como cubrirlas de la manera más óptima se llega a la conclusión que tiene que tener la siguiente estructura:

Tabla 4: *Topics* planteados para abarcar los requisitos funcionales.

	Concepto	Nombre del topic	Tipos de Datos
Topics	**Comandar**	/command	motor_msgs/cmd_motor
	Mover a posición inicial	/default_position	std_msgs/Empty

Tabla 5: Servicios planteados para abarcar los requisitos funcionales.

	Concepto	Nombre del Topic	Tipos de Datos
Servicios	**Calibrar**	/calibrate	motor_msgs/Calibrate
	Activar/Desactivar	/disable	motor_msgs/Enable
	Obtener estado	/get_status	motor_msgs/GetState
	Ping	/ping	motor_msgs/Ping

Y en cuanto a los datos que se quieren enviar o recibir, se plantean los siguientes para cada tipo:

Tabla 6: Datos para cada mensaje empleado en la API.

Tipo de Dato	Descripción del Tipo de Dato
std_msgs/Empty [/default_position]	
motor_msgs/Enable [disable]	bool data (falso habilitar, verdadero deshabilitar) string name (nombre del motor) ---
motor_msgs/Ping [/ping]	string name (nombre del motor) --- bool success string reason (razón en caso de éxito o error)
motor_msgs/GetState [/get_state]	string name --- int32[] id float64 position float64 velocity float64 acceleration float64 torque float64 goal float64 voltage int32[] temperature bool is_moving float64 error
motor_msgs/cmd_motor [/command]	float32 acceleration float32 velocity float32 position uint8 ABS = 0 # absolute position uint8 REL = 1 # relative position uint8 type String name

Con el planteamiento de la solución realizado, ya se puede empezar el **desarrollo de la solución** de la API. Para ello se quiere utilizar un solo nodo encargado de gestionar todas las funcionalidades. Pero antes se necesitan crear dos paquetes: uno que contenga el nodo mencionado *(uc3m_dynamixel_controller)* y otro que contenga los mensajes y servicios que se van a utilizar *(motor_msgs)*.

4.2.2.1 Paquete motor_msgs

Primero se necesita crear el paquete en el cual se encuentran todos los mensajes y servicios que se van a utilizar. A este paquete se le da el nombre de *motor_msgs*[2]. La estructura del paquete es la siguiente:

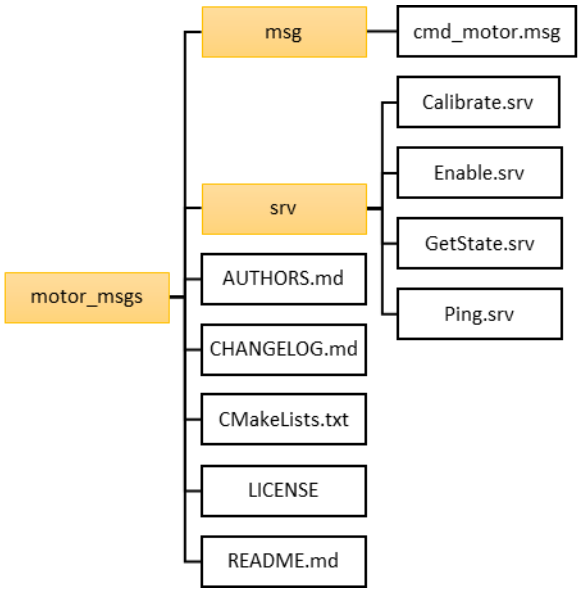

Figura 14: Estructura del paquete *motor_msgs*.

Los datos de cada uno de los documentos de mensajes y servicios son los indicados en la tabla 4 y 5. Con este paquete creado se comienza a programar la API.

4.2.2.2 Paquete uc3m_dynamixel_controller

El paquete *uc3m_dynamixel_controller*, va a ser el que contenga el nodo principal gestor de la API. La estructura de esta carpeta se muestra en la figura 14. A parte de las carpetas contenidas dentro de este paquete, las cuales se explican con detenimiento a continuación presenta 2 documentos: *CMakeList.txt* (en el cual se escriben las

[2] Este paquete ha sido creado por el Laboratorio de Robótica Social de la UC3M. En concreto, por Álvaro y Miguel Ángel Salisch. Aunque, este diseño primero, se ha modificado para adaptarlo a la API creada.

directivas, los ejecutables, las librerías, los mensajes, los servicios, etc,) y *package.xml* (en el cual se escriben las dependencias del paquete).

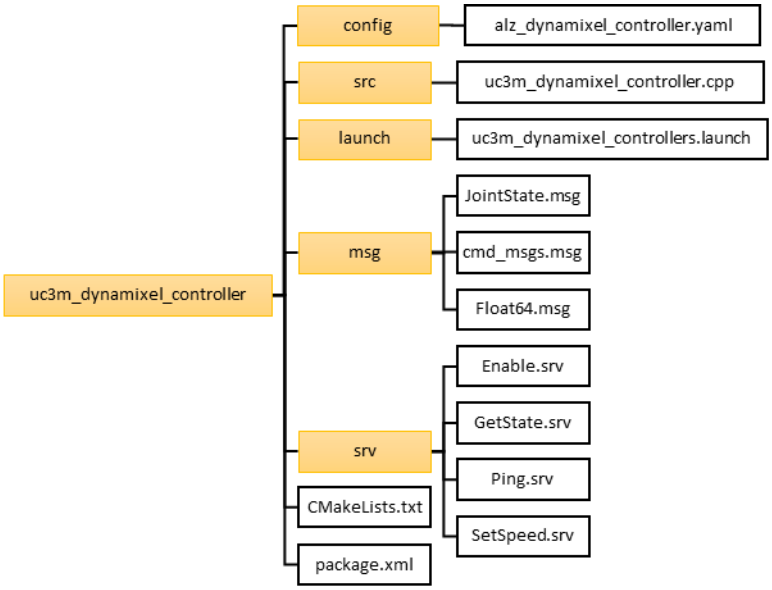

Figura 15: Estructura del paquete *uc3m_dynamixel_controller*.

4.2.2.3 Carpeta config

Esta carpeta es esencial para el funcionamiento de los motores. Tiene que incluir un documento *.yaml*, en el cual se insertan los parámetros necesarios para la inicialización de los motores. En nuestro caso este recibe el nombre de *alz_dynamixel_controller.yaml*. (véase al completo en el anexo 1). A continuación, se muestra y se explica la configuración del motor base:

```
base:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: base
    joint_speed: 1.6
    motor:
        id: 1
        init: 512
        min: 256
        max: 768
```

- controller: dentro de este apartado hay que introducir los comandos indicados para que el motor funcione en el modo “joint” o unión, y no en el modo “Wheel” o rueda. De esta manera, el motor se comporta como una articulación y no como una rueda.
- joint_name: parámetro que indica el nombre que se quiere dar al motor.
- joint_speed: inicializa la velocidad del motor con el valor que se le introduzca, en unidades de radianes por segundo [rad/s].
- motor:
  - id: se introduce el valor del id que se le quiere dar al motor. Se le puede dar cualquier valor. En caso de tener varios motores el valor tiene que ser único.
  - init, min y max: estos parámetros indican la posición inicial, el límite inferior y el límite superior de movimiento del motor. Los valores van de 0 hasta 1023; siendo el 0 igual a 0 grados centígrados, y siendo 1023 igual a 300 grados centígrados (véase la imagen XX). Por lo que se puede deducir que hay una zona sobre la cual no se puede mover el motor. Debido a la posición del motor en la cabeza del robot, en este caso se le da un valor de inicio de 512, es decir 150º y unos valores límites de 256 y 768, aproximadamente unos ±75 grados respecto a la posición inicial.

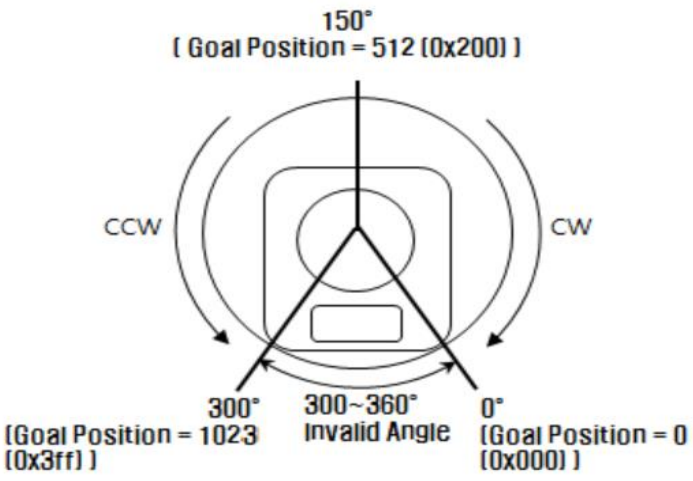

Figura 16: Esquema del movimiento del motor dynamixel AX-12. [1]

4.2.2.4 Carpeta src

Como bien saben los usuarios de ROS, la capeta *src* contiene los nodos con el código fuente. Aquí se programará la API en el documento *uc3m_dynamixel_controller.cpp*. Este documento es el nodo principal y único de la API. Para consultar el código al completo véase el anexo 2. A continuación, se irá explicando la programación del nodo por las distintas secciones de las que se forma.

- **Encabezamiento**: se incluyen las librerías y los documentos de ficheros necesarios para la realización de la API, las constantes y las variables globales empleadas.

```
#include <ros/ros.h>
#include <motor_msgs/cmd_motor.h>
#include <motor_msgs/Enable.h>
#include <motor_msgs/GetState.h>
#include <motor_msgs/Ping.h>
#include <std_msgs/Empty.h>
#include <dynamixel_msgs/JointState.h>
#include <std_msgs/Float64.h>
#include <dynamixel_controllers/SetSpeed.h>
#include <dynamixel_controllers/TorqueEnable.h>

#define RADIAN 57.3

ros::Publisher pos1, pos2, pos3, pos4, pos5;
```

A continuación, se comenta la utilización de cada uno de los elementos:

```
#include <ros/ros.h>
```

Se trata de una librería que contiene los elementos necesarios de ROS para publicar, subscribirse, enviar *topics*, etc.

```
#include <motor_msgs/cmd_motor.h>
#include <motor_msgs/Enable.h>
#include <motor_msgs/GetState.h>
#include <motor_msgs/Ping.h>
#include <std_msgs/Empty.h>
```

Estos documentos son los mensajes creados por el alumno, y que recibe la API para cada una de las acciones necesarias: mover un motor, habilitar el motor, obtener el estado, etc.

```
#include <dynamixel_msgs/JointState.h>
#include <std_msgs/Float64.h>
#include <dynamixel_controllers/SetSpeed.h>
#include <dynamixel_controllers/TorqueEnable.h>
```

Estos cuatro son los empleados por el paquete *dynamixel_motor* (mencionado en el capítulo 3.3) para obtener los parámetros de los motores, mover el motor, configurar la velocidad y habilitar el par.

```
#define RADIAN 57.2958
```

Se declara una constante para la conversión de radianes a grados, que se realizará en numerosos casos de la API.

```
ros::Publisher pos1, pos2, pos3, pos4, pos5;
```

Declaración de los *publishers*, encargados del movimiento de cada uno de los motores.

- **Main:** en el main de la API, se inicializa el nodo "*uc3m_dynamixel_controller*", los *publishers* encargados del movimiento de cada uno de los motores, y se subscribe a las 5 funcionalidades desarrolladas: 2 topics,"*/command*"(encargado del movimiento de los motores) y "*/default_position*"(encargado de mover todos los motores a la posición inicial) y 3 servicios, "*/get_status*"(obtiene los parámetros de un motor determinado), "*/enable*"(permite el par del motor por una fuerza externa) y "*/ping*"(verifica que el motor este operativo). Cada uno de los callbacks se ejecutará cuando el programa llegue al *ros::spin()*.

```
int main(int argc, char** argv){

    ros::init(argc, argv, "uc3m_dynamixel_controller");
```

```
    ros::NodeHandle nh;


    pos1=nh.advertise<std_msgs::Float64>("/base/command",0);
    pos2=nh.advertise<std_msgs::Float64>("/rightArm/command",0);
    pos3=nh.advertise<std_msgs::Float64>("/leftArm/command",0);
    pos4=nh.advertise<std_msgs::Float64>("/neck/command",0);
    pos5=nh.advertise<std_msgs::Float64>("/head/command",0);


    ros::Subscriber command=nh.subscribe("/command",10,command_callback);
    ros::Subscriber
def_pos=nh.subscribe("/default_position",10,default_pos_callback);


    ros::ServiceServer srv_dis = nh.advertiseService("/enable",enable_callback);
    ros::ServiceServer srv_st =
nh.advertiseService("/get_status",state_callback);
    ros::ServiceServer srv_pi = nh.advertiseService("/ping",ping_callback);


    ros::spin();


    return 0;
}
```

- ***Callbacks*:** a continuación, se comenta el desarrollo de cada uno de los *callbacks* utilizados, cada uno con su propia función:
    - ***command_callback*:** encargado del movimiento de un motor. Primero, se expone el *callback* al completo, y luego se analiza paso por paso.

```
    void command_callback(const motor_msgs::cmd_motor::ConstPtr &msg){


    dynamixel_controllers::SetSpeed sp;
    std_msgs::Float64 pos;


    std::string joint=msg->name;
    std::ostringstream joint_state;
    joint_state << "/" << joint << "/state";


    float curr_pos;


    if(msg->type==0){
        pos.data=(msg->position)/RADIAN;
        if((joint=="leftArm")||(joint=="head")||(joint=="neck")){
```

```
            pos.data=-(msg->position)/RADIAN;
        }
    }else if(msg->type==1){
        dynamixel_msgs::JointStateConstPtr motor_msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
        curr_pos=(motor_msg->current_pos)*RADIAN;
        pos.data=((msg->position)+curr_pos)/RADIAN;
        if((joint=="leftArm")||(joint=="head")||(joint=="neck")){
            pos.data=(-(msg->position)+curr_pos)/RADIAN;
        }

    }else{
        ROS_INFO("Incorrect value for type. 0 for ABSOLUTE, 1 for RELATIVE
commanding");
    }

    sp.request.speed=msg->velocity;
    float acc=msg->acceleration;

    std::ostringstream service_call;
    service_call << "/" << joint << "/set_speed";
    ros::NodeHandle nh_speed;
    ros::ServiceClient
speed=nh_speed.serviceClient<dynamixel_controllers::SetSpeed>(service_call.st
r());

    ros::Rate loop_rate(10);
    float curr_pos2, goal_pos, error;
    int cont=0;

       do{
            speed.call(sp);

            if(cont<1){
                if(joint=="base")
                    pos1.publish(pos);
                else if(joint=="rightArm")
                    pos2.publish(pos);
                else if(joint=="leftArm")
                    pos3.publish(pos);
                else if(joint=="neck")
                    pos4.publish(pos);
```

```
                else if(joint=="head")
                    pos5.publish(pos);
                else
                    ROS_INFO("Introduce a correct motor name");
            }


            sp.request.speed=sp.request.speed+(msg->acceleration)/10;


            dynamixel_msgs::JointStateConstPtr msg2=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
            curr_pos2=msg2->current_pos*RADIAN;
            goal_pos=msg2->goal_pos*RADIAN;


            error=goal_pos-curr_pos2;


            loop_rate.sleep();


            cont++;


        }while((error>1)||(error<(-1)));//end loop
}
```

```
 void command_callback(const motor_msgs::cmd_motor::ConstPtr &msg)
```

Comienzo del *callback* "*command_callback*", el cual recibe un mensaje de tipo *motor_msgs::cmd_motor*, con los datos mostrados en la tabla 6, que se reciben a través de la variable msg.

```
   dynamixel_controllers::SetSpeed sp;
   std_msgs::Float64 pos;


   std::string joint=msg->name;
   std::ostringstream joint_state;
   joint_state << "/" << joint << "/state";
```

Se declaran las variables necesarias para realizar el movimiento de los motores. La variable *sp*, de tipo *dynamixel_controllers::SetSpeed*, es del tipo de mensaje encargado de fijar la velocidad del motor y la variable *pos* de tipo *std_msgs::Float64*, es del tipo de mensaje encargado de fijar la posición. Ambas variables son las que se enviarán al nodo principal del paquete

dynamixel_controller para fijar velocidades, darle aceleración al movimiento y enviar la posición a la cual se quiere hacer el movimiento.

Por otro lado, se realiza una lectura del nombre del motor sobre el cual se desea realizar la operación y se crea una variable de tipo *ostringstream* la cual la es rellenada. Este *ostringstream* es el nombre del *topic* emisor del estado del motor (envía datos de posición, velocidad, temperatura, etc.) y se utilizará para subscribirse a este *topic* y recibir así la posición del motor previa a un movimiento para realizar movimientos relativos y absolutos.

```
float curr_pos;


    if(msg->type==0){
        pos.data=(msg->position)/RADIAN;
        if((joint=="leftArm")||(joint=="head")||(joint=="neck"))
            pos.data=-(msg->position)/RADIAN;
    }else if(msg->type==1){
        dynamixel_msgs::JointStateConstPtr motor_msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
        curr_pos=(motor_msg->current_pos)*RADIAN;
        pos.data=((msg->position)+curr_pos)/RADIAN;
        if((joint=="leftArm")||(joint=="head")||(joint=="neck")){
            pos.data=(-(msg->position)+curr_pos)/RADIAN;
        }
    }else{
        ROS_INFO("Incorrect value for type. 0 for ABSOLUTE, 1 for RELATIVE
commanding");
    }
```

Esta parte del código es la encargada de guardar la posición a la cual se tiene que mover el motor, en función de si se quiere realizar un movimiento absoluto o relativo. Para saber si el movimiento tiene que ser absoluto, se recibe una variable llamada *type*. Si se recibe un "0", el movimiento será absoluto, y si se recibe un "1" relativo.

Para ello, primero se declara un variable *float* llamada *curr_pos*, en la cual se guardará más adelante la posición actual del motor. Después, se comienza con la selección de lectura de información de posición en función del tipo movimiento. Primero, se realiza una estructura condicional (*if)* a la cual se

entra en caso de tratarse de un movimiento absoluto. Aquí dentro se transforman los datos de grados a radianes, ya que el envío de posición al nodo *dynamixel_manager* (encargado del movimiento del motor), es en radianes y se guardan en la variable *pos*. Debido a la orientación de los motores "*leftArm*", "*head*" y "*neck*", para realizar el movimiento en el sentido según en el que hemos establecido nuestro sistema de referencia, se necesita cambiarles el signo a estos tres motores.

En el caso de recibir un "1", es decir posición relativa, entramos en la siguiente estructura condicional. Lo primero que se hace es subscribirse al *topic* que envía datos de los motores, en concreto del motor del cual queremos realizar el movimiento. Para ello, se utiliza la herramienta "*ros::topic::waitForMessage*"[3]. Esta herramienta recibe un mensaje asociado a un topic, lo guarda en una variable y termina. Este mensaje se guarda en este caso en la variable "*motor_msg*", la cual tiene que ser del mismo tipo que el topic recibido (*dynamixel_msgs::JointState*). A continuación, se guarda la posición actual del motor, contenida en "*motor_msgs.current_pos*" (en radianes), en *curr_pos* (transformada a grados). Se realiza una operación de suma para que el movimiento relativo sea posible, y se cambia de signo en caso de tratarse de los tres motores mencionados anteriormente.

Y por último, en el caso de que no se reciba ni un "0", ni un "1", se imprime por pantalla un *ROS_INFO*.

```
    sp.request.speed=msg->velocity;
    float acc=msg->acceleration;


    std::ostringstream service_call;
    service_call << "/" << joint << "/set_speed";
    ros::NodeHandle nh_speed;
```

[3] Esta herramienta tiene una gran ventaja en el caso de nuestro código, ya que se subscribe a un *topic* tan solo una vez. En este caso el *topic* al que se subscribe esta publicando a una frecuencia de 100Hz, que en el caso de querer subscribirnos de una manera convencional daría lugar a un programa mucho menos eficiente, que la solución planteada.

```
    ros::ServiceClient
speed=nh_speed.serviceClient<dynamixel_controllers::SetSpeed>(service_call.s
tr());
```

Una vez leída la posición a la cual se desea mover el motor y lista para enviar, se procede a leer la velocidad y la aceleración recibidas en el mensaje. Debido a que la velocidad, tal y como se ha comentado antes, se envía a través de un servicio, se rellena una variable con el nombre del servicio encargado de esta función y se declara el servicio "*speed*" que enviará más adelante la velocidad a la que se quiere ejecutar el movimiento.

```
    ros::Rate loop_rate(10);
    float curr_pos2, goal_pos, error;
    int cont=0;


       do{
            speed.call(sp);


            if(cont<1){
                if(joint=="base")
                    pos1.publish(pos);
                else if(joint=="rightArm")
                    pos2.publish(pos);
                else if(joint=="leftArm")
                    pos3.publish(pos);
                else if(joint=="neck")
                    pos4.publish(pos);
                else if(joint=="head")
                    pos5.publish(pos);
                else
                    ROS_INFO("Introduce correct motor name");
            }


            sp.request.speed=sp.request.speed+(msg->acceleration)/10;


            dynamixel_msgs::JointStateConstPtr msg2=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
            curr_pos2=msg2->current_pos*RADIAN;
            goal_pos=msg2->goal_pos*RADIAN;


            error=goal_pos-curr_pos2;
```

```
            loop_rate.sleep();

            cont++;

        }while((error>1)||(error<(-1)));//end loop
}
```

Por último, tras haber realizado toda la lectura de datos, conversiones y modificaciones necesarias, se procede a enviar los datos. Para esto, primero utilizamos la herramienta "*ros::Rate loop_rate()"*, que permite realizar una parada en la ejecución del programa, en función del valor introducido.

Se entra en el bucle "*do-while*" y lo primero es enviar la velocidad deseada por servicio. Después se procede a enviar la posición, que se enviará tan solo una vez por uno u otro publisher (ya declarados e inicializados como variables globales y en el main) en función del motor a mover.

El único valor que nos falta por enviar es la aceleración. Ya que no existe un parámetro de aceleración para este tipo de motor, debido a que no existe en el nodo *dynamixel_manager*, ni en ninguno de los registros de los motores, se utiliza la ecuación de velocidad para un movimiento curvilíneo uniformemente acelerado (1), que se encargará de aportar aceleración al motor. Esto se consigue actualizando la velocidad 10 veces por segundo[4] mediante la ecuación mencionada, la cual se enviará al principio de cada repetición del bucle dando lugar a una aceleración en el motor.

$$v = v_0 + a \cdot t \quad (1)$$

Para acabar, es necesario saber cuándo ha llegado el motor a su posición con el fin de terminar la actualización de la aceleración y el *callback*. Por ello, se realiza una lectura de la posición a la cual se desea mover el motor. Cuando se publica la posición a la cual se desea mover un motor se fija esta posición en el *topic* que envía el estado del motor. Debido a esto, se vuelve a realizar una subscripción a este *topic* con la misma herramienta que antes, se hace una

[4] Se elije esta magnitud para que no sea muy grande como para consumir más CPU y no sea lo bastante pequeña para que el cambio de velocidad no sea apreciable.

lectura de la posición a la que se desea mover y de la posición actual. Ambas posiciones se restan para saber el espacio que queda hasta alcanzar esa posición. En el momento que el valor entre ambas posiciones sea menor que una unidad se termina el bucle.

- ***default_pos_callback*:** este *callback*, está encargado de devolver todos los motores a su posición inicial. Al igual que en el anterior *callback*, primero se muestra el código al completo, y después se analiza paso por paso.

```
void default_pos_callback(const std_msgs::Empty::ConstPtr& msg){

    std::vector<std::string> joints;
    joints.push_back("/base");
    joints.push_back("/rightArm");
    joints.push_back("/leftArm");
    joints.push_back("/neck");
    joints.push_back("/head");

    dynamixel_controllers::SetSpeed sp;
    sp.request.speed=1;

    std_msgs::Float64 pos;
    pos.data=0;

    ros::NodeHandle nh_position;
    ros::NodeHandle nh_speed;

    ros::Rate loop_rate(10);

    float posi;

    for(int i=0;i<joints.size();i++){

        std::string joint_name(joints[i]);
        std::ostringstream topic_name, topic_state, service_call;
        topic_name << joint_name << "/command";
        service_call << joint_name << "/set_speed";
        topic_state << joint_name << "/state";
```

```
        ros::Publisher
position=nh_position.advertise<std_msgs::Float64>(topic_name.str(),0);

        ros::ServiceClient
speed=nh_speed.serviceClient<dynamixel_controllers::SetSpeed>(service_call.str(
));


        dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(topic_state.str());

        posi=msg->current_pos*RADIAN;


        if(posi>1){

            position.publish(pos);

            std::string x= topic_name.str();

            ROS_INFO("Publish %s, to position %f",x.c_str(),pos.data);

            speed.call(sp);

        }else if(posi<(-1)){

            position.publish(pos);

            std::string x= topic_name.str();

            ROS_INFO("Publish %s, to position %f",x.c_str(),pos.data);

            speed.call(sp);

        }

        loop_rate.sleep();

    }

}
```

```
void default_pos_callback(const std_msgs::Empty::ConstPtr& msg)
```

Comienzo del *callback* "*default_pos_callback*", el cual recibe un mensaje vacío (*std_msgs::Empty*).

```
    std::vector<std::string> joints;

    joints.push_back("/base");

    joints.push_back("/rightArm");

    joints.push_back("/leftArm");

    joints.push_back("/neck");

    joints.push_back("/head");
```

Se inicializa y se rellena un vector con los nombres de los cinco vectores, que se utilizarán más adelante.

```
    std_msgs::Float64 pos;
```

```
    pos.data=0;


    dynamixel_controllers::SetSpeed sp;
    sp.request.speed=1;


    ros::NodeHandle nh_position;
    ros::NodeHandle nh_speed;
```

Igualmente, se inicializa la variable que se encargarán de enviar los motores a la posición inicial, es decir 0. Además, se crea una variable para el envío de la velocidad con valor unitario con el fin de que los movimientos de todos los motores sean a una velocidad baja e igual, en el caso de que sus velocidades tuvieran un valor mucho mayor y dos *NodeHandle* para las declaraciones de envío de *topic* (para la posición) y de servicio (para la velocidad).

```
    ros::Rate loop_rate(10);
    float posi;


    for(int i=0;i<joints.size();i++){


        std::string joint_name(joints[i]);
        std::ostringstream topic_name, topic_state, service_call;
        topic_name << joint_name << "/command";
        service_call << joint_name << "/set_speed";
        topic_state << joint_name << "/state";


        ros::Publisher
position=nh_position.advertise<std_msgs::Float64>(topic_name.str(),0);
        ros::ServiceClient
speed=nh_speed.serviceClient<dynamixel_controllers::SetSpeed>(service_call.str(
));


        dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(topic_state.str());
        posi=msg->current_pos*RADIAN;


        if(posi>1){
            speed.call(sp);
           position.publish(pos);
            std::string x= topic_name.str();
            ROS_INFO("Publish %s, to position %f",x.c_str(),pos.data);
        }else if(posi<(-1)){
```

```
            speed.call(sp);
            position.publish(pos);
            std::string x= topic_name.str();
            ROS_INFO("Publish %s, to position %f",x.c_str(),pos.data);
        }
        loop_rate.sleep();
    }
```

Por último, en este *callback*, se realiza el bucle que enviará la posición inicial a los motores que no estén en ella.

Para ello, primero se declara fuera del bucle la cantidad de segundos de espera, para que se realice el bucle y una variable llamada "*posi*", que utilizaremos para leer la posición actual de cada uno de los motores.

Con el comienzo del bucle, que queda delimitado por el tamaño del vector donde se han inicializado el nombre de los motores, se va guardando en tres *ostringstream* distintos los nombre de cada uno de los *topics* con los que se van a realizar alguna operación para cada motor. Después se declara el *publisher*[5](que enviará la posición), el servicio (que enviará la velocidad) y realizamos una subscripción con la herramienta *ros::topic::waitForMessage* para leer la posición del motor. Una vez sabido esto, se verifica que la posición del motor del cual se trate (depende del número de iteración en la que se encuentre el bucle) sea mayor y menor que 1 ya que los motores no siempre se encuentran en 0 cuando están en su posición inicial, sino que pueden estar a unas décimas de grado. Por ello se mueven los motores que estén a una posición mayor o menor que 1 y no a una posición distinta de 0. En caso de que no esté en lo que se considera como posición inicial, se fija la velocidad del motor a mover a 1 radián por segundo y se envía la posición.

- ***disable_callback*:** este *callback* se encarga de habilitar y deshabilitar la aplicación de par al motor por parte de una fuerza exterior.

[5] En *default_pos_callback* la declaración del *publisher* se realiza de forma distinta a la del *command_callback*, con el fin de optimizar el código y de proponer una solución distinta a la ya realizada.

```
bool enable_callback(motor_msgs::Enable::Request& req,
motor_msgs::Enable::Response& res){



    std::string joint=req.name;


    bool en=req.data;

    dynamixel_controllers::TorqueEnable torque;

    torque.request.torque_enable=en;


    std::ostringstream service_call;

    service_call << "/" << joint << "/torque_enable";

    ros::NodeHandle nh_enable;

    ros::ServiceClient
enable=nh_enable.serviceClient<dynamixel_controllers::TorqueEnable>(service_cal
l.str());


    enable.call(torque);


    return true;

}
```

Debido a la existencia de esta funcionalidad ofrecida por el nodo *dynamixel_manager* del paquete *dynamixel_controller*, este callback es simple. Se ha implantado en la API con el fin de que las principales funcionalidades queden aquí compactas.

El servicio existente para esta función se llama "*/motor_id/torque_enable*", donde *motor_id* es el nombre de cada motor. Por ello, se recibe el nombre del motor, y se crea el nombre del servicio al cual se va a llamar, y mediante una variable booleana se indica si se quiere habilitar o deshabilitar ("false" para habilitar y "true" para deshabilitar). Una vez realizado lo anterior, se envía el servicio para que se realice la operación.

- ***state_callback*:** este *callback* tiene como objetivo la lectura de los parámetros de los motores.

```
bool state_callback(motor_msgs::GetState::Request& req,
motor_msgs::GetState::Response& res){


    std::string joint=req.name;
```

```
    std::ostringstream joint_state;
    joint_state << "/" << joint << "/state";

    dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
    float pos, goal_pos, error;
    pos=msg->current_pos*RADIAN;
    goal_pos=msg->goal_pos*RADIAN;
    error=msg->error*RADIAN;
    if((joint=="base")||(joint=="rightArm")){
        res.position=pos;
        res.goal=goal_pos;
        res.error=error;
    }else{
        res.position=-(pos);
        res.goal=-(goal_pos);
        res.error=-(error);
    }
    res.is_moving=msg->is_moving;
    res.voltage=msg->load;
    res.id=msg->motor_ids;
    res.temperature=msg->motor_temps;
    res.velocity=msg->velocity;

    return true;
}
```

Este otro *callback*, también existe en el nodo *dynamixel_manager* del paquete *dynamixel_controllers*, pero los datos de posición se muestran en radianes y con un signo distinto al que se toma en el trabajo como referencia para algunos motores.

Para ello, se subscribe en el *callback* al *topic* que envía los datos de los motores, mediante la misma herramienta que se ha utilizado a lo largo del nodo para la lectura de estos datos. Se convierten los datos a grados y se cambia el signo; estos se guardan en la respuesta del servicio que se ha creado para realizar esta lectura y se finaliza el *callback*.

- ***ping_callback*:** este *callback*, se encarga de verificar que el motor se encuentra operativo e inicializado.

```
bool ping_callback(motor_msgs::Ping::Request& req, motor_msgs::Ping::Response&
res){


    std::vector<int> ping;


    std::string joint=req.name;

    std::ostringstream joint_state;

    joint_state << "/" << joint << "/state";


    dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());

    ping=msg->motor_ids; //Conversion from radians to degrees


    if((ping[0]==0)&&(!ping[0])){

        res.success=0;

        res.reason="Motor disconnected";

    }else{

        res.success=1;

        res.reason="Motor connected";

    }

    return true;

}
```

Cuando se lanza el *launcher* (se explicará más adelante) por completo, donde se incluye el nodo *uc3m_dynamixel_controller*, *dynamixel_manager* y el documento *.yaml* de configuración, los motores se inicializan con un id propio, el cual le da el desarrollador en el documento de configuración. De este modo el *callback* verificará si el parámetro *id*, uno de los que envía el *topic* de estado de los motores; si se verifica que es distinto de 0, este se encontrará operativo.

Para ello, se recibe la petición del servicio con el nombre del motor sobre el cual se desea realizar esta verificación. Se realiza una subscripción igual que en los *callbacks* anteriores, y se lee el ping del motor en concreto, el cual se guarda en un vector de tipo *int* (mismo tipo que el dato que se lee). En el caso que ping este vacío o sea 0, se envía un mensaje mediante la respuesta del servicio indicando que el motor se encuentra desconectado. Si es 1, se envía un mensaje indicando que el motor está conectado y operativo.

4.2.2.5 Carpeta launch

Otra de las carpetas importantes en los paquetes de ROS, son las *launch*, las cuales contienen los documentos de tipo *launch*. Estos documentos son necesarios para poder ejecutar dos o más nodos y presentan un formato XML. Para lanzar el documento *launch* y poder ejecutar lo necesario para hacer funcionar la API, necesitamos incluir el nodo *uc3m_dynamixel_controller* (que contiene toda la API anteriormente explicada), cargar el documento de configuración llamado *alz_dynamixel_controller.yaml* (documento donde se establecen los parámetros de configuración de los motores) y el nodo *dynamixel_manager*(encargado de gestionar los *topics* y servicios para realizar acciones con los motores). A parte de estos tres elementos se debe incluir en el documento el nodo *tilt_contorller_spawner*, nodo también incluido en el paquete *dynamixel_controllers*, encargado de inicializar los motores. Quedando el documento *launch* de la siguiente manera:

```
<launch>


     <rosparam command="load" file="$(find
uc3m_dynamixel_controller)/config/alz_position_controllers.yaml"/>


     <node name="tilt_controller_spawner" pkg="dynamixel_controllers"
type="controller_spawner.py"
          args="--manager=dxl_manager
                --port pan_tilt_port
                base rightArm leftArm neck head"
          output="screen"/>


     <node name="dynamixel_manager" pkg="dynamixel_controllers"
type="controller_manager.py" respawn="true" output="screen">
     <rosparam>
                namespace: dxl_manager
                serial_ports:
                    pan_tilt_port:
                        port_name: "/dev/servos"
                        baud_rate: 1000000
                        min_motor_id: 1
                        max_motor_id: 25
                        update_rate: 20
       </rosparam>
     </node>
```

```
    <node name = "uc3m_dynamixel_controller"
            pkg = "uc3m_dynamixel_controller"
            type = "uc3m_dynamixel_controller"
            output="screen"
            respawn= "true"/>
</launch>
```

4.2.2.6 Carpeta msg

Esta carpeta, contiene la declaración de los mensajes que se utilizan en el nodo *uc3m_dynamixel_controller*, y es necesario incluirlas en esta carpeta para que se puedan realizar acciones con ellos en este nodo. Los mensajes utilizados son los siguientes:

- **Cmd_motor.msg**: mensaje que contiene los datos que se envían al nodo *uc3m_dynamixel_controllers* para mover el motor, a ciertas velocidades y aceleraciones. Contenido en la carpeta *motor_msgs*. Compuesto por los siguientes datos:

```
# Description of standard ROS message to control any motor.
# It allows to set the acceleration, deceleration, velocity, and target position.

# header info
 Header header

# Sets the acceleration/deceleration of the motor.
# A positive value changes the acceleration.
# If the value is negative, it sets the deceleration.
# acceleration [=] radians/sec^2
float32 acceleration

# Sets the velocity of the motor.
# velocity [=] radians/sec
float32 velocity

# This values sets the goal position of the motor.
# A positive value moves it forward, whereas a negative one moves it backwards.
# A 0 value does not specify the direction.
# position [=] degrees
float32 position
```

```
# This values sets the torque of the motor.
float32 torque


# Depending on the type, the movement will be relative to the current position or
absolute.
# The default value is ABS.
uint8 ABS = 0 # absolute position
uint8 REL = 1 # relative position
uint8 type


#Name of the motor
string name
```

- **JointState.msg:** mensaje contenido en la carpeta *dynamixel_controllers*, y que es el mensaje del *topic* que publica el nodo *dynamixel_manager* con los parámetros de posición, velocidad, temperatura, etc. Presenta los siguientes datos:

```
Header header
string name         # joint name
int32[] motor_ids   # motor ids controlling this joint
int32[] motor_temps # motor temperatures, same order as motor_ids

float64 goal_pos    # commanded position (in radians)
float64 current_pos # current joint position (in radians)
float64 error       # error between commanded and current positions (in radians)
float64 velocity    # current joint speed (in radians per second)
float64 load        # current load
bool is_moving      # is joint currently in motion
```

- **Float64.msg:** mensaje contenido en la carpeta *std_msgs* de ROS. Este mensaje, es el que envía el topic "/*motor_id/command*" al nodo *dynamixel_manager*, encargado de mover el motor. En nuestro caso, se utiliza para enviarlo desde el nodo *uc3m_dynamixel_controller*, al *dynamixel_manager*.

```
float64 data
```

4.2.2.7 Carpeta srv

La carpeta *srv*, contiene la declaración de los servicios que se utilizan en el nodo *uc3m_dynamixel_controller*. Al igual que con los mensajes, es necesario incluirlos en

esta carpeta, para que se puedan realizar acciones con ellos en este nodo. Los mensajes utilizados son los siguientes:

- **Enable.srv:** este servicio, se encuentra en la carpeta *motor_msgs*. Se crea para enviar la información al nodo *uc3m_dynamixel_controller* sobre si se quiere habilitar o deshabilitar el motor y el nombre del motor sobre el que se desea realizar la operación. Por ello, presenta los siguientes datos:

```
string name # motor name
bool data # motor enabling / disabling
---
```

- **GetState.srv:** servicio, que se encuentra también en la carpeta *motor_msgs*. Se crea, con la finalidad de enviar al nodo *uc3m_dynamixel_controller,* el nombre del motor sobre el que se desea recibir información de su estado. Esta información, es recibida primero, a través de un *topic* con el mensaje *JointState.msg* (descrito antes), que lo envía *dynamixel_manager*. Hay que adecuar los datos a unas unidades y signos deseados.

```
string name                 # name of the motor you want to read status
---
int32[] id                              # unique id of the motor
float64 position              # current position of the motor in degrees
float64 velocity              # current velocity of the motor in rad/sec
float64 acceleration # current acceleration of the motor in rad/sec^2
float64 torque          # torque of the motor
float64 goal            # goal position of the motor in degrees
float64 voltage         # current voltage of the motor
int32[] temperature     # current temperature of the motor
bool is_moving          # 1-motor is moving 0-motor isnt moving
float64 error           # current position - goal position
```

- **Ping.srv:** servicio utilizado para verificar si el motor, se encuentra o no operativo. Recibe el nombre del motor deseado, y devuelve un mensaje sobre el éxito de la operación. Formado por:

```
string name          #name of the motor you want to check
---
bool success
```

```
string reason
```

- **SetSpeed.srv:** este último servicio, se encuentra en el paquete *dynamixel_motor* y es el servicio que recibe *dynamixel_manager* para cambiar la velocidad de movimiento de los motores. Está compuesto por:

```
float64 speed
---
```

5 RESULTADOS

En este capítulo, se realiza un análisis de los resultados obtenidos, a partir de la solución planteada en el capítulo anterior. Se realizará un análisis de los resultados funcionales (requisitos básicos para la realización de la API), como de requisitos no funcionales, igualmente importantes a la hora de realizar este trabajo, tal y como se comentó en el inicio del capítulo 4.

5.1 Resultados funcionales

En este apartado se demuestra el cumplimiento de los requisitos funcionales mencionados en la tabla 3. Para demostrarlos, se muestran mediante imágenes de secuencias de movimiento e información obtenida a través de terminal, los resultados obtenidos de las distintas funcionalidades:

1. **Comandar:** funcionalidad encargada del movimiento del robot. Primero, se exponen los resultados para el movimiento absoluto de los motores y después los resultados para el movimiento relativo:

   - **Movimiento absoluto:** en las imágenes se muestra el resultado del envío de la instrucción de movimiento absoluto (figura 16). Con la figura 17, se verifica que efectivamente que el nodo uc3m_dynamixel_controller envía el *topic /leftArm/command* encargado del movimiento del brazo izquierdo. Después de las imágenes de terminal, se muestra la secuencia de movimiento que desempeña el motor con la aplicación de la instrucción de la figura 16.

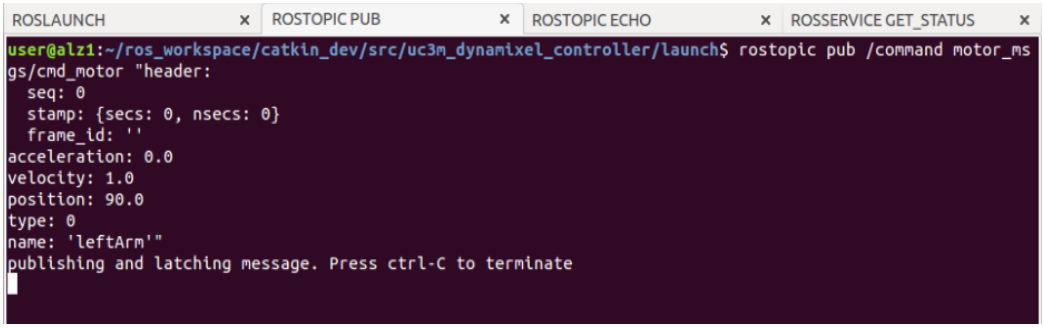

Figura 17: Instrucción de movimiento del brazo izquierdo a 90º y 1rad/s.

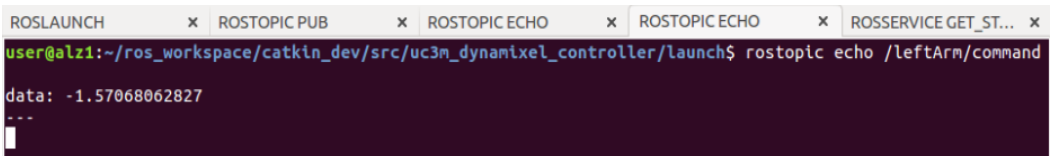

Figura 18: Rostopic echo de /leftArm/command.

Figura 21: Robot Mini en posición inicial.

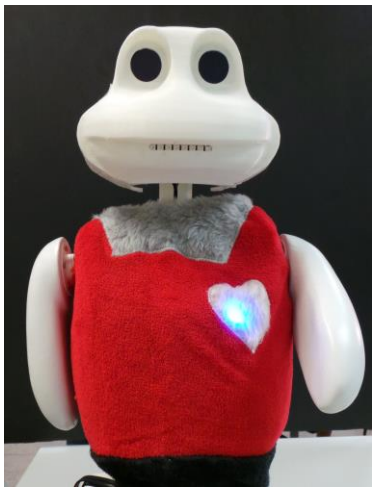

Figura 19: Robot Mini con brazo izquierdo en movimiento.

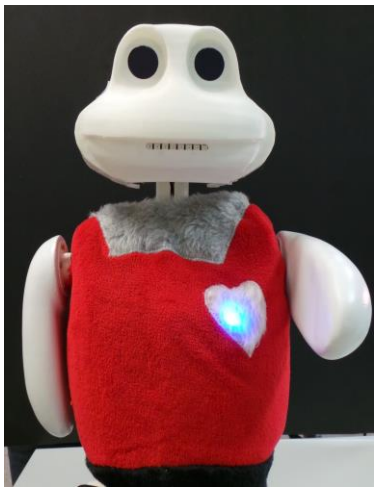

Figura 20: Robot Mini con brazo izquierdo a 90.

- **Movimiento relativo:** en las siguientes imágenes, se muestra como la instrucción de mover el brazo derecho a una posición relativa de -45 grados (partiendo el brazo de 90 grados), introducida por pantalla, envía el *topic /rightArm/command* a la posición de 45 grados (figura 22), es decir, 45 grados menos de su posición actual, que es 90.

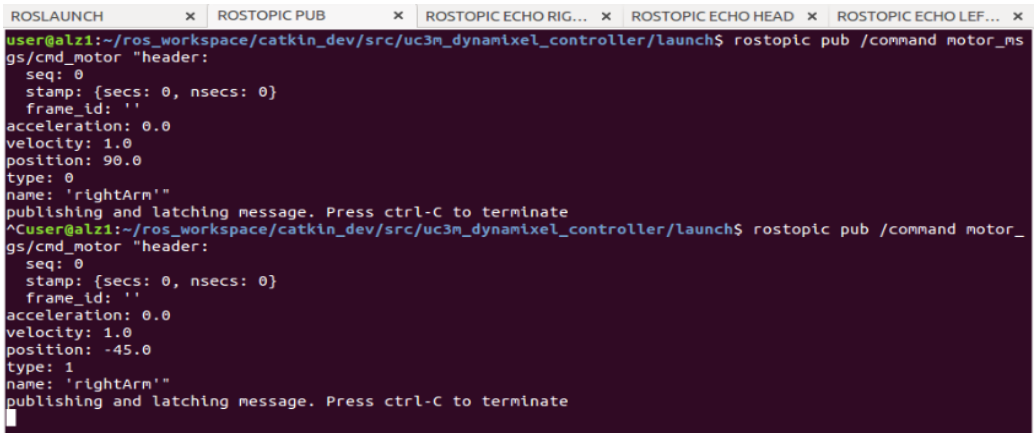

Figura 22: Instrucción de movimiento relativo del brazo derecho.

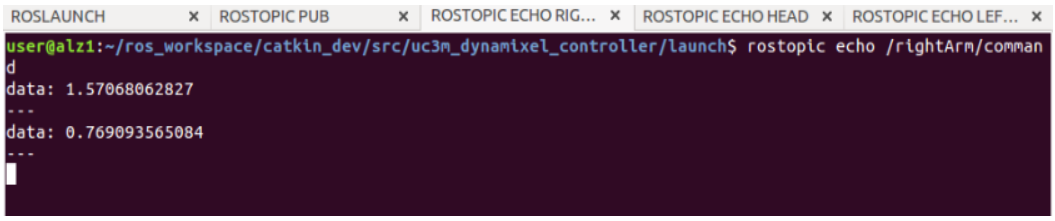

Figura 23: *Rostopic echo,* del topic */rightArm/command.*

2. **Mover motores a posición inicial:** esta función está encargada de mover los motores que no se encuentren en su posición de inicio a ella. Para demostrarlo, se muestra la instrucción introducida por pantalla de la función mover a posición inicial (figura 21). Después de haber movido los motores del brazo izquierdo, derecho y cabeza a posiciones distintas a la inicial. En las figuras 22, 23 y 24 se muestra la secuencia de movimiento después de ejecutar la instrucción de mover los motores a posición inicial.

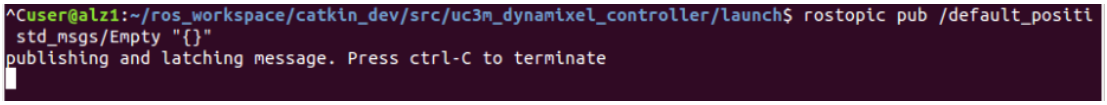

Figura 24: Instrucción mover motores a posición inicial.

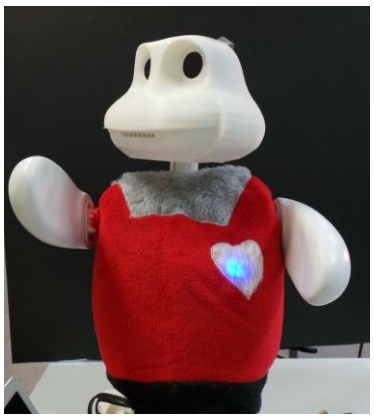

Figura 27: Robot Mini con motores brazo izquierdo, derecho y cabeza en posiciones distintas a la inicial.

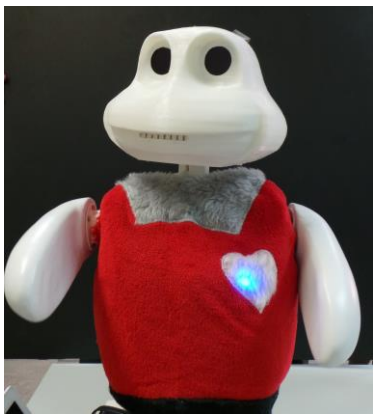

Figura 26: Robot Mini volviendo a posición inicial.

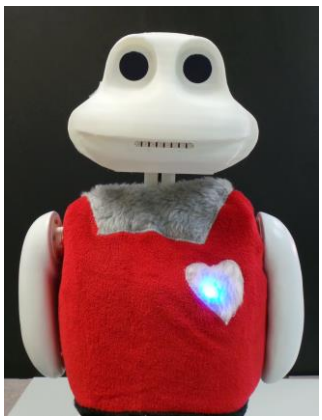

Figura 25: Robot Mini en posición inicial.

3. **Obtener estado del motor:** esta función se encarga de obtener el valor de determinados parámetros de los distintos motores. En la figura 23 se muestra los valores que se obtienen del servicio */get_status* (encargado de esta función) para el brazo derecho, cuando este se encuentra a 45 grados.

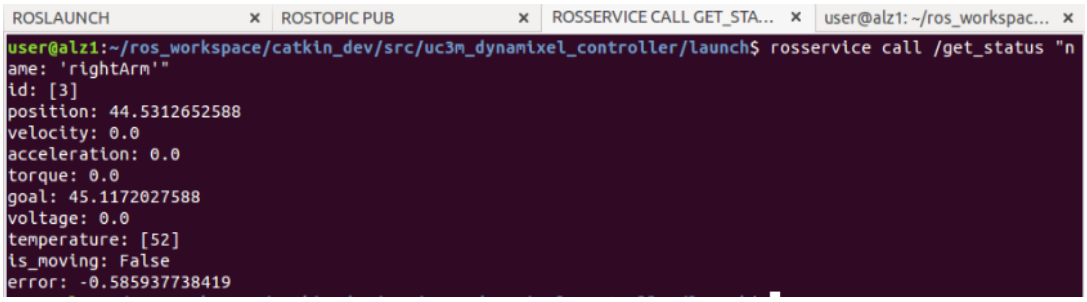

Figura 28: Imagen de lo que se obtiene en terminar al lanzar el servicio */get_status.*

4. **Ping:** está función verifica que el motor se encuentra activo. Se muestra a continuación el ejemplo para el motor del brazo derecho:

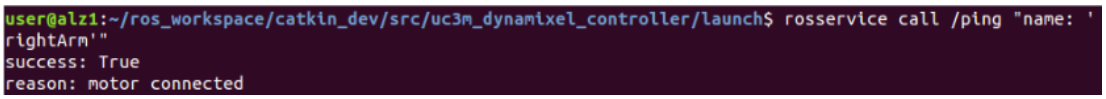

Figura 29: Servicio */ping* para el brazo derecho del robot.

5. **Activar/Desactivar:** esta función permite habilitar el movimiento por parte de una fuerza externa y deshabilitarlo. Ya que no devuelve nada por pantalla al enviarse la instrucción no se adjunta imagen.

5.2 Resultados no funcionales

En este apartado se muestra el consumo de CPU (véase la figura 16) y de memoria de la API.

Primero, mediante el uso del comando *"htop"*, se verifica que el consumo de memoria por parte de la API es aproximadamente de un 0,4% del total de memoria.

Por otro lado, en la tabla se refleja el consumo de CPU del nodo *uc3m_dynamixel_controller*, que contiene todo el código fuente de la API y el consumo de Python, que es el nodo *dynamixel_manager*, encargado de la gestión de los motores.

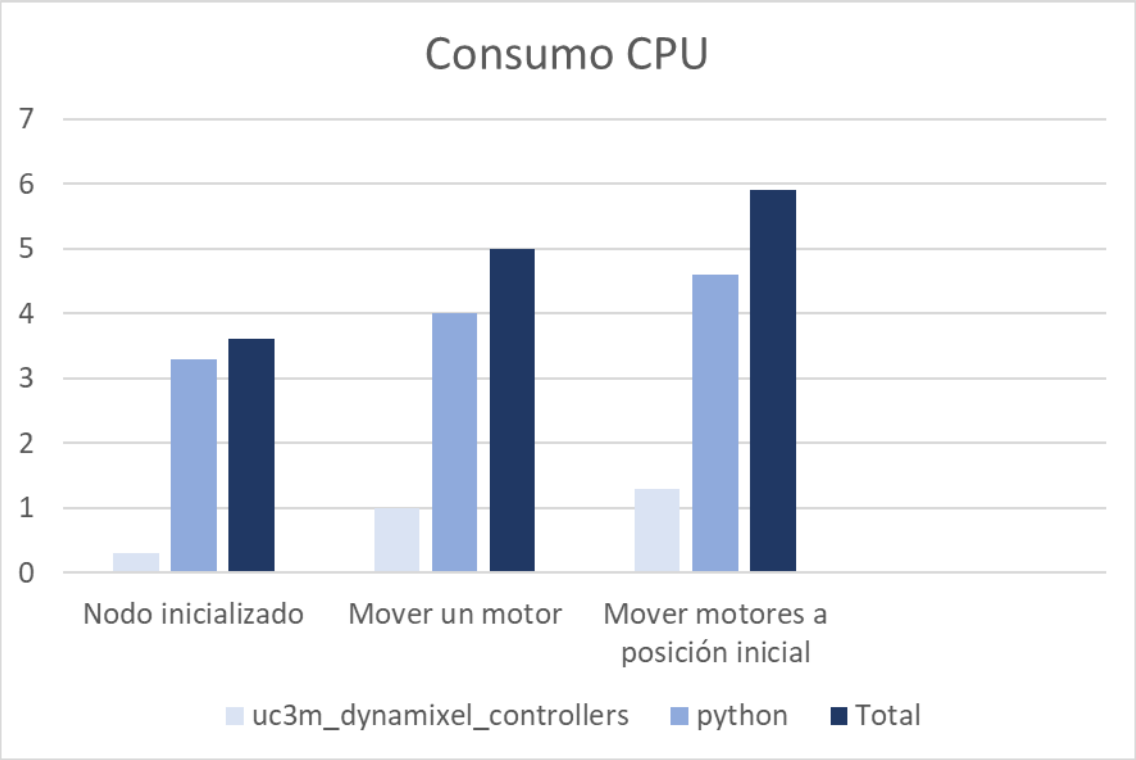

Figura 30: Gráfica del consumo de CPU de la API.

6 CONCLUSIONES

Con la realización de este trabajo, se ha conseguido realizar una API accesible, funcional y robusta para incorporar al Proyecto Mini del Laboratorio de Robótica Social de la Universidad Carlos III. Esta nueva API de control de los motores existentes, permitirá al robot Mini adquirir una mayor naturalidad en sus gestos. Esta característica es fundamental, tal y como se comentó en la introducción del trabajo, en los robots sociales, donde cuanto más próximo y parecido sea al ser humano, mejor será el robot social.

A parte de la consecución de los requisitos funcionales, es necesario el cumplimiento de unos requisitos no funcionales. En proyectos como este, donde esta API no es ni un 1% de todo lo que es el proyecto Mini, es necesario que el consumo de CPU y memoria durante su funcionamiento sea mínimo, para que pueda trabajar a la vez con el resto de APIs. Se deduce de la tabla 16 que el consumo de CPU de la API (uc3m_dynamixel_controllers) oscila entre 0,3% y un 1,3%. El nodo encargado del movimiento de los motores (Python) introducido en el capítulo 3.3 consume hasta 4,6% de CPU. Sumando ambos nodos obtenemos un valor mínimo de consumo de 3,6% y un valor máximo de 5,9%. Al tratarse del grueso que gestiona todos los motores, se puede decir que son valores muy razonables. Por ello, aparte de haber realizado el trabajo de una manera eficaz, se ha conseguido de una forma eficiente, alcanzando un bajo consumo de CPU y memoria.

Por último, como conclusión personal, cabe destacar que el alumno partía desde el completo desconocimiento del funcionamiento del campo de la robótica, en concreto de ROS. Para conseguir el correcto funcionamiento del trabajo, han sido necesarias muchas horas de estudio de la materia de manera autodidacta y la resolución de muchos problemas que aparecen en el camino; desde la incapacidad de interpretar el problema a resolver, hasta problemas de compilación que han durado semanas. Esto ha supuesto un gran reto personal y ha permitido al alumno alcanzar un amplio conocimiento y aumentar su inquietud en esta materia.

7 LÍNEAS DE TRABAJO FUTURAS

A continuación, se desarrollan las líneas de trabajo futuras que se pueden aplicar a este trabajo:

- **Ampliación y mejora de las funcionalidades existentes:** la API desarrollada cubre las funcionalidades mencionadas en la tabla 3, debido a las necesidades actuales del robot Mini. A medida que vayan cambiando las necesidades en el movimiento del robot, la API estará sujeta cambios y ampliaciones de código. Entre las posibilidades existentes están:
    - **Realización de gestos:** al igual que en la API realizada existe una función que devuelve todos los motores a su posición inicial, se pueden desarrollar funciones que desarrollen gestos y movimientos específicos que se utilicen con frecuencia en el robot.
    - **Funciones de control:** la API es capaz de leer la temperatura y voltaje de los motores. Con el fin de asegurar un buen uso de los motores y evitar accidentes, se pueden implantar funciones encargadas de cerciorar que las temperaturas y voltajes de los motores se encuentren dentro de un rango óptimo. Y en el caso de que no se encuentren dentro del rango, desactivarlos de la fuente de alimentación.
- **Cambio de elección del motor:** los motores empleados (Dynamixel AX-12), son servos de gama media alta con un precio elevado para tratarse de un robot prototipo de laboratorio de universidad. En el mercado existen numerosos motores de menor precio que ofrecen características similares, aunque una menor calidad, pero aun así, capaces de cubrir las funcionalidades descritas.

  Otro motivo de elección de otro motor, puede ser la necesidad de una funcionalidad, que el motor Dynamixel AX-12 sea incapaz de realizar. Ya que este, tiene unas funcionalidades limitadas; aquellas que se encuentran en cada uno de los registros de la RAM y EEPROM.

En cuanto a las futuras líneas de trabajo más allá de este proyecto, en el desarrollo de movimientos en robots sociales, hay todavía mucho por hacer. El objetivo final, como se ha repetido numerosas veces en el documento, es que el movimiento sea lo más

parecido al del ser humano. Por ello, todavía queda mucho por avanzar en cuanto a la calidad del movimiento. Pero un factor muy importante y que desnaturaliza por completo los movimientos de los robots, es el ruido que producen al moverse. Hay que investigar y mejorar el aislamiento del sonido que generan los pequeños motores utilizados en la robótica social.

8 MANUAL DE USO

En este capítulo, se repasan las funcionalidades de la API y el modo de uso por parte del usuario, desde el proceso de instalación, hasta el proceso de ejecución de las distintas funcionalidades[6].

8.1 Instalación

Primero, antes de nada, es necesario instalar el paquete *dynamixel_motor* del cual se parte y con el cual se comunica la API creada. Para ello hay que seguir los siguientes pasos de instalación. [3]

1. Instalar el paquete *dynamixel_motor* del repositorio.

```
sudo apt-get install ros-%ROS_DISTRO%-dynamixel-motor


sudo apt-get install ros-indigo-dynamixel-motor
```

   En este ejemplo se instala para ROS Indigo, en el caso de ser otra versión de ROS, introducir el nombre de la versión.

2. Se puede revisar la última versión del repositorio con el siguiente comando:

```
git clone https://github.com/arebgun/dynamixel_motor.git
```

Una vez instalada y revisada la última versión de *dynamixel_motor*, es necesario implementar la API desarrollada: el paquete *motor_msgs* y *uc3m_dynamixel_controller*.

8.2 Pasos previos

El primer paso para poder utilizar la API, consiste en ejecutar el *launcher* "*uc3m_dynamixel_controllers.launch*". Para ello, el usuario tiene que hacer lo siguiente:

1. Abrir un terminal y situarse dentro del paquete *uc3m_dynamixel_controllers* en la carpeta *launch.*
2. Una vez dentro de la carpeta *launch*, lanzar el documento *launch*, ejecutando en el terminal:

[6] Recuerda que tabulando se introducen los comandos mucho más rápidamente que escribiéndolos al completo en el teclado.

```
$ roslaunch uc3m_dynamixel_controller.launch
```

3. Se espera a que se ejecute el *launcher* y se abre una nueva ventana en el terminal para comenzar su uso.

8.3 Funcionalidades y uso

La API, ofrece las funcionalidades mostradas en la tabla 6. Debido, a que algunas funcionalidades se utilizan a través de *topics* y otras a través de servicios, se hablará primero de las funcionalidades basadas en *topics* y después en las basadas en servicios.

Pero antes, se muestra una tabla donde quedan resumidas las unidades de todos los parámetros y variables utilizadas en la API.

Tabla 7: Unidades de cada uno de los parámetros involucrados en la API.

Parámetros y Variables	Unidades
Position	Grados [º]
Goal_position	Grados [º]
Velocity	Radianes por segundo [rad/s]
Acceleration	Radianes por segundo al cuadrado [rad/s^2]
Temperature	Grados centígrados [℃]
Voltage	Voltios [V]
Error	Grados [º]

8.3.1 Uso de los topics

El número de funcionalidades que utilizan una comunicación a través de *topics* son dos:

- ***/command***: tal y como se ha mencionado anteriormente este *topic* es el encargado del movimiento de los motores. Para publicar el *topic* ejecutamos el siguiente comando:

```
$ rostopic pub /command motor_msgs/cmd_motor
```

Una vez introducido esto, tabulando nos aparecerá el siguiente texto (para el caso del motor de la cabeza)

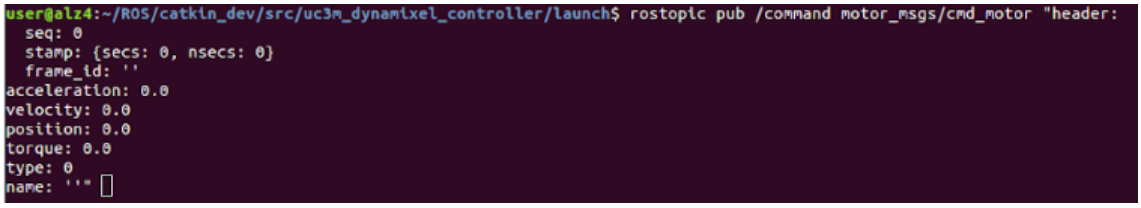

Figura 31: Parámetros a introducir por terminal para mover un motor.

- name: introduce el nombre del motor que se desea mover. Depende del nombre que le hayas dado en el documento *config*. En este caso las únicas posibilidades son: *head, neck, leftArm, rightArm, base*.
- acceleration: introduce la aceleración con la que desea el usuario mover el motor en rad/s^2.
- velocity: introduce la velocidad de movimiento del motor en rad/s.
- position: unidades en grados. Si se desea mover el motor hacia delante, introducir valores positivos. Para mover el motor hacia atrás, introducir valores negativos.
- type: hay dos opciones que permiten realizar dos tipos de movimientos.
  - "0": realiza movimientos absolutos; es decir, se mueve a la posición que introduzcas en la variable position.
  - "1": realiza movimientos relativos; es decir; se mueve las unidades introducidas en la variable position respecto a la posición que presenta el motor en ese instante.

- ***/default_position***: se trata de un *topic* vacío, encargado de devolver los motores a su posición inicial. Para hacerlo, simplemente hay que introducir el siguiente comando:

```
$ rostopic pub /default_position std_msgs/Empty "{}"
```

8.3.2 Uso de servicios

Entre los servicios que ofrece la API se encuentran los siguientes:

- ***/get_status***: este servicio, nos permite obtener los datos de un motor. En este servicio, hay que introducir tan sólo un dato de entrada, el nombre del motor del que se desea obtener información. Se ejecuta de la siguiente forma:

```
$ rosservice call /get_status motor_msgs/Getstate name “”
```

Los parámetros que recibimos a continuación son los siguientes, para el caso del motor de la cabeza (véase figura 18):

```
user@alz4:~/ROS/catkin_dev/src/uc3m_dynamixel_controller/launch$ rosservice call /get_status "name: 'head'"
id: [5]
position: -0.0
velocity: 0.0
acceleration: 0.0
torque: 0.0
goal: -0.0
voltage: 0.0
temperature: [37]
is_moving: False
error: -0.0
```

Figura 32: Parámetros que se reciben por teminal de un motor (en este caso del motor *head*).

- ***/ping***: este servicio, también tiene el mismo parámetro de entrada que *get_status*; el nombre del motor del que interesa saber si se encuentra o no operativo. Para realizarlo, se ejecuta el siguiente comando:

```
$ rosservice call /ping motor_msgs/Ping name “”
```

Como respuesta de nuestro servicio, se obtiene un parámetro booleano y un mensaje, donde se indica si el motor está o no operativo.

```
user@alz4:~/ROS/catkin_dev/src/uc3m_dynamixel_controller/launch$ rosservice call /ping "name: 'head'"
success: True
reason: motor connected
```

Figura 33: Respuesta del servicio */ping* mostrada por el terminal.

- ***/motor_id/torque_enable***: este servicio, permite habilitar o deshabilitar el movimiento del motor por parte de una fuerza externa. En el nombre del servicio, se indica el nombre del motor del cual se quiere habilitar o deshabilitar su movimiento

Para habilitarlo, se introduce lo siguiente (ejemplo para el motor base):

```
$ rosservice call /base/torque_enable dynamixel_msgs/torque_enable “false”
```

Para deshabilitar el movimiento:

```
$ rosservice call /base/torque_enable dynamixel_msgs/torque_enable “true
```

BIBLIOGRAFÍA

[1] «AX-12/AX-12+/AX-12A». [En línea]. Disponible en: http://support.robotis.com/en/product/actuator/dynamixel/ax_series/dxl_ax_actuator.htm. [Accedido: 22-sep-2018].

[2] «User´s Manual. Dynamixel AX-12». ROBOTIS. 2006-06-14. [En línea]. Disponible en: http://www.crustcrawler.com/products/bioloid/docs/AX-12.pdf. [Accedido: 22-sep-2018]

[3] «dynamixel_motor - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/dynamixel_motor. [Accedido: 22-sep-2018].

[4] A. J. P. Vidal y F. Alonso-Martın, «Evolución de la robótica social y nuevas tendencias», p. 9.

[5] «Figure 1. El Robot Social Mini (Fotografía Original Tomada El 25 De...» ResearchGate. Accedido 25 de septiembre de 2018. https://www.researchgate.net/figure/El-robot-social-Mini-fotografia-original-tomada-el-25-de-Mayo-de-2017-en-el-laboratorio_fig1_321333117.

[6] L. Olavarrieta, J. Ramón, P. Coronel, y S. Orellana Pérez, «Historia, evolución, estado actual y futuro de la cirugía robótica», *Revista de la Facultad de Medicina*, vol. 30, n.º 2, pp. 109-114, dic. 2007.

[7] «Jibo Robot Voice Command Smart Assistant and Speaker - 8747696 | HSN». Accedido 25 de septiembre de 2018. https://www.hsn.com/products/jibo-robot-voice-command-smart-assistant-and-speaker/8747696.

[8] L. O. de Málaga, «La robótica humanoide, una eficaz herramienta terapéutica». [En línea]. Disponible en: https://www.laopiniondemalaga.es/sociedad/2016/07/11/robotica-humanoide-dotes-sociales-eficaz/862848.html. [Accedido: 21-sep-2018].

[9] «Leonardo da Vinci's robots - LEONARDO3 - Leonardo da Vinci | BOOKS - LIBRI». [En línea]. Disponible en: http://www.leonardo3.net/leonardo/books%20I%20robot%20di%20Leonardo%20-%20Taddei%20Mario%20-%20english%20Leonardo%20robots%202.html#5. [Accedido: 22-sep-2018].

[10] «La robótica ASIMO Futuro de Alta tecnología - Caminar Robot png dibujo - Transparente png dibujo Juguete png Descargar.» Accedido 25 de septiembre de 2018. https://es.kisspng.com/kisspng-zrn76r/.

[11] QUIGLEY, Morgan, et al. ROS: an open-source Robot Operating System. En *ICRA workshop on open source software*. 2009. p. 5.

[12] «Tipos de motores rotativos para proyectos de Arduino», *Luis Llamas*.

[13] M. A. Salichs, M. Malfaz, y J. F. Gorostiza, «Toma de Decisiones en Robótica», *Revista Iberoamericana de Automática e Informática Industrial RIAI*, vol. 7, n.º 4, pp. 5-16, oct. 2010.

LISTA DE ACRÓNIMOS

UC3M = Universidad Carlos III de Madrid

TFG = Trabajo de Fin de Grado

SDK = Software Development Kit (Kit de desarrollo de software)

STAIR = Stanford Artificial Intelligence Robot

API = Application Programming Interface

EEPROM = Electrically Erasable Programmable Read-Only Memory

RAM = Random Access Memory

Third-Party= Desarrollador a terceros

ANEXO 1: CÓDIGO DOCUMENTO DE CONFIGURACIÓN (alz_dynamixel_controller.yalm)

```
base:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: base
    joint_speed: 1.6
    motor:
        id: 1
        init: 512
        min: 256
        max: 768


leftArm:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: leftArm
    joint_speed: 1.3
    motor:
        id: 2
        init: 512
        min: 0
        max: 639


rightArm:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: rightArm
    joint_speed: 1.3
    motor:
        id: 3
        init: 512
        min: 384
```

```
        max: 1023


neck:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: neck
    joint_speed: 1.3
    motor:
        id: 4
        init: 512
        min: 295
        max: 735


head:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: head
    joint_speed: 1.3
    motor:
        id: 5
        init: 512
        min: 354
        max: 620
```

ANEXO 2: CÓDIGO API (uc3m_dynamixel_controller.cpp)

```
#include <ros/ros.h>
#include <motor_msgs/cmd_motor.h>
#include <motor_msgs/Enable.h>
#include <motor_msgs/GetState.h>
#include <motor_msgs/Ping.h>
#include <motor_msgs/Calibrate.h>
#include <dynamixel_msgs/JointState.h>
#include <std_msgs/Float64.h>
#include <std_msgs/Empty.h>
#include <dynamixel_controllers/SetSpeed.h>
#include <dynamixel_controllers/TorqueEnable.h>

#define RADIAN 57.2958

//JointState parameter
dynamixel_msgs::JointState _joint_state;
//Declaration publishers for commanding
ros::Publisher pos1, pos2, pos3, pos4, pos5;

////////////////////////////////////////////////////////////////////////////////
void command_callback(const motor_msgs::cmd_motor::ConstPtr &msg){

    //Parameters used to set position and speed
    dynamixel_controllers::SetSpeed sp;
    std_msgs::Float64 pos;

    //Read joint name
    std::string joint=msg->name;
    //Create string to subscribe to /motor_id/state
    std::ostringstream joint_state;
    joint_state << "/" << joint << "/state";

    //Variable use to read current position for relative commanding
    float curr_pos;

    //Absolute and Relative position commanding
    if(msg->type==0){
        //ABSOLUTE
        pos.data=(msg->position)/RADIAN;
```

```
        //leftArm neck head sign change
        if((joint=="leftArm")||(joint=="head")||(joint=="neck")){
            pos.data=-(msg->position)/RADIAN;
        }
    }else if(msg->type==1){
        //RELATIVE
        //Subscribe to /motor_id/state to receive current position
        dynamixel_msgs::JointStateConstPtr motor_msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
        curr_pos=(motor_msg->current_pos)*RADIAN;
        pos.data=((msg->position)+curr_pos)/RADIAN;
        //leftArm neck head sign change
        if((joint=="leftArm")||(joint=="head")||(joint=="neck")){
            pos.data=(-(msg->position)+curr_pos)/RADIAN;
        }

    }else{
        ROS_INFO("Incorrect value for type. 0 for ABSOLUTE, 1 for RELATIVE commanding");
    }

    //Read velocity acceleration
    sp.request.speed=msg->velocity;
    float acc=msg->acceleration;

    //Create a service client to send the speed value
    //Speed is set at the servo by service /motor_id/set_speed
    std::ostringstream service_call;
    service_call << "/" << joint << "/set_speed";
    ros::NodeHandle nh_speed;
    ros::ServiceClient
speed=nh_speed.serviceClient<dynamixel_controllers::SetSpeed>(service_call.str());

    ros::Rate loop_rate(10);
    float curr_pos2, goal_pos, error;
    int cont=0;

        //Starts loop till motor in goal_pos
        do{
            //Send speed service
            speed.call(sp);
```

```
            //Send position
            if(cont<1){
                if(joint=="base")
                    pos1.publish(pos);
                else if(joint=="rightArm")
                    pos2.publish(pos);
                else if(joint=="leftArm")
                    pos3.publish(pos);
                else if(joint=="neck")
                    pos4.publish(pos);
                else if(joint=="head")
                    pos5.publish(pos);
                else
                    ROS_INFO("Introduce a correct motor name");
            }

            //Acceleration ecuation, upgrades speed 10 times per second
            sp.request.speed=sp.request.speed+(msg->acceleration)/10;

            //Verify if position can be reached with data introduced
            dynamixel_msgs::JointStateConstPtr msg2=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
            curr_pos2=msg2->current_pos*RADIAN;
            goal_pos=msg2->goal_pos*RADIAN;
            //ROS_INFO("Current pos %f", curr_pos2);

            error=goal_pos-curr_pos2;

            loop_rate.sleep();

            cont++;

            //ROS_INFO("Error %f", error);
        }while((error>1)||(error<(-1)));//end loop
}

////////////////////////////////////////////////////////////////////////////////
void default_pos_callback(const std_msgs::Empty::ConstPtr& msg){

    //Vector declaration and filling with names of the joints
```

```
    std::vector<std::string> joints;
    joints.push_back("/base");
    joints.push_back("/rightArm");
    joints.push_back("/leftArm");
    joints.push_back("/neck");
    joints.push_back("/head");
    //Parameter used to set a low speed to return to default position in case it was
high
    dynamixel_controllers::SetSpeed sp;
    //Low speed set
    sp.request.speed=1;

    //Parameter used to set the initial position
    std_msgs::Float64 pos;
    //Initial position set
    pos.data=0;

    ros::NodeHandle nh_position;
    ros::NodeHandle nh_speed;



    ros::Rate loop_rate(10);

    //Parameter used to verify if pos set to 0
    float posi;

    //Move all joints to default position
    for(int i=0;i<joints.size();i++){



        ROS_INFO("Comienzo bucle, vuelta %i",i);
        std::string joint_name(joints[i]);
        std::ostringstream topic_name, topic_state, service_call;
        topic_name << joint_name << "/command";
        service_call << joint_name << "/set_speed";
        topic_state << joint_name << "/state";


        ros::Publisher
position=nh_position.advertise<std_msgs::Float64>(topic_name.str(),0);
        ros::ServiceClient
speed=nh_speed.serviceClient<dynamixel_controllers::SetSpeed>(service_call.str());
```

```
        //Subscribe to /motor_id/state to obtain data from the motor_id
        dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(topic_state.str());
        posi=msg->current_pos*RADIAN;

        if(posi>1){
            position.publish(pos);
            std::string x= topic_name.str();
            ROS_INFO("Publish %s, to position %f",x.c_str(),pos.data);
            speed.call(sp);
        }else if(posi<(-1)){
            position.publish(pos);
            std::string x= topic_name.str();
            ROS_INFO("Publish %s, to position %f",x.c_str(),pos.data);
            speed.call(sp);
        }

        loop_rate.sleep();
    }
    //ROS_INFO("Saliendo default_pos_callback");
}

////////////////////////////////////////////////////////////////////////////////

bool enable_callback(motor_msgs::Enable::Request& req, motor_msgs::Enable::Response&
res){

    //Read joint wanted to enable
    std::string joint=req.name;
    //Read action wanted (false->enable, true->disable)
    bool en=req.data;
    dynamixel_controllers::TorqueEnable torque;
    torque.request.torque_enable=en;
    //Create a service client to send action wanted
    //Torque enable or disable is done by service /motor_id/torque_enable
    std::ostringstream service_call;
    service_call << "/" << joint << "/torque_enable";
    ros::NodeHandle nh_enable;
    ros::ServiceClient
enable=nh_enable.serviceClient<dynamixel_controllers::TorqueEnable>(service_call.str());

    enable.call(torque);
```

```
    return true;
}

////////////////////////////////////////////////////////////////////////////////

bool state_callback(motor_msgs::GetState::Request& req, motor_msgs::GetState::Response&
res){

    //Read joint name wanted to read value
    std::string joint=req.name;
    std::ostringstream joint_state;
    joint_state << "/" << joint << "/state";

    //Subscribe to /motor_id/state to obtain data from the motor_id
    dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());
    float pos, goal_pos, error;
    pos=msg->current_pos*RADIAN;
    goal_pos=msg->goal_pos*RADIAN;
    error=msg->error*RADIAN;
    if((joint=="base")||(joint=="rightArm")){
        res.position=pos;
        res.goal=goal_pos;
        res.error=error;
    }else{
        res.position=-(pos);
        res.goal=-(goal_pos);
        res.error=-(error);
    }
    res.is_moving=msg->is_moving;
    res.voltage=msg->load;
    res.id=msg->motor_ids;
    res.temperature=msg->motor_temps;
    res.velocity=msg->velocity;

    return true;
}

////////////////////////////////////////////////////////////////////////////////
```

```
bool ping_callback(motor_msgs::Ping::Request& req, motor_msgs::Ping::Response& res){


    std::vector<int> ping;


    std::string joint=req.name;

    std::ostringstream joint_state;

    joint_state << "/" << joint << "/state";

    //Verify if the motor has an id

    dynamixel_msgs::JointStateConstPtr msg=
ros::topic::waitForMessage<dynamixel_msgs::JointState>(joint_state.str());

    ping=msg->motor_ids; //Conversion from radians to degrees


    if((ping[0]==0)||(!ping[0]){

        res.success=0;

        res.reason="motor disconnected";

    }else{

        res.success=1;

        res.reason="motor connected";

    }


    return true;

}


////////////////////////////////////////////////////////////////////////////////////

/// \brief main

/// \param argc

/// \param argv

/// \return

///

int main(int argc, char** argv){


    //Initialise node uc3m_dynamixel_controller

    ros::init(argc, argv, "uc3m_dynamixel_controller");


    ros::NodeHandle nh;


    //Initialisation of command topics

    pos1=nh.advertise<std_msgs::Float64>("/base/command",0);

    pos2=nh.advertise<std_msgs::Float64>("/rightArm/command",0);

    pos3=nh.advertise<std_msgs::Float64>("/leftArm/command",0);

    pos4=nh.advertise<std_msgs::Float64>("/neck/command",0);
```

```
    pos5=nh.advertise<std_msgs::Float64>("/head/command",0);


    //subscribe to topics:
    // 1) /command
    // 2) /default_position

    ros::Subscriber command=nh.subscribe("/command",10,command_callback);
    ros::Subscriber def_pos=nh.subscribe("/default_position",10,default_pos_callback);

    //subscribe to service:
    // 1) /enable
    // 2) /get_status
    // 3) /ping

    ros::ServiceServer srv_dis = nh.advertiseService("/enable",enable_callback);
    ros::ServiceServer srv_st = nh.advertiseService("/get_status",state_callback);
    ros::ServiceServer srv_pi = nh.advertiseService("/ping",ping_callback);



    ros::spin();

    return 0;
}
```